MASTER EN SCIENCES INFORMATIQUES

Vers un analyseur statique générique de Java par interprétation abstraite: un utilitaire d'affichage de l'environnement et du store

Pirotte, Cécile

Award date:
2001

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Vers un analyseur statique générique de Java par interprétation abstraite :

un utilitaire d'affichage de l'environnement et du store

Cécile PIROTTE

Mémoire présenté pour l'obtention du grade de maître en Informatique

Année académique 2000 - 2001

RÉSUMÉ

L'analyse statique désigne l'ensemble des traitements qui peuvent être appliqués à un programme en dehors de son exécution. Sa mise en œuvre a pour but d'obtenir des informations permettant l'optimisation et la vérification de programmes.

L'interprétation abstraite est un cadre mathématique général pour l'analyse statique. Elle permet d'exprimer les méthodes d'analyse statique existantes et de prouver leur correction. Le principe de base de l'interprétation abstraite est l'exécution du programme à analyser sur un domaine non standard appelé domaine abstrait. Le domaine abstrait est un ensemble d'éléments dont les valeurs concrètes présentent des caractéristiques communes. Ces éléments doivent ainsi représenter des propriétés du domaine standard valables pour toutes les exécutions possibles du programme.

L'objectif du projet JavAbInt est de réaliser un analyseur statique générique de Java par interprétation abstraite. La première étape du projet a été franchie avec la définition de la syntaxe et de la sémantique opérationnelle d'un sous-langage de Java, ainsi qu'avec la mise en œuvre de domaines abstraits pour ce langage par I. Pollet [POLLET]. Deux mémoires ont également été réalisés dans le cadre de ce projet.

Le but de ce travail est le développement d'un utilitaire d'affichage du couple composé de l'environnement et du store, aussi bien au niveau du domaine abstrait que du domaine concret. Le store pouvant être représenté par un graphe et l'environnement pouvant y être intégré sous la forme de sommets particuliers, cet utilitaire consiste principalement en la réalisation d'une application permettant l'affichage d'un graphe quelconque. Ceci nous amène ainsi à nous intéresser aux techniques de dessin de graphe.

ABSTRACT

Static analysis refers to all the treatments which can be applied to a program without executing it The results produced can then be used to optimize and verify programs.

Abstract interpretation is a general mathematical framework for static analysis. It allows us to formalize the existing methods of static analysis and to prove their correction. Abstract interpretation is based on the execution of the program to be analyzed on a nonstandard domain called the abstract domain. The abstract domain is a set of elements whose concrete values share some properties. These elements should thus represent valid properties of the standard domain for all the possible executions of the program.

The objective of project JavAbInt is to realize a generic static analyzer of Java based on abstract interpretation. The first part of the project consisted in defining the syntax and the operational semantics of an sublanguage of Java, as well as in the definition of abstract domains for this language by I. Pollet [POLLET]. Two theses were also realized within the framework of this project.

The aim of this work is to develop an utility which displays the environment and the store, for both the abstract and the concrete domains. As the store can be represented by a graph and the environment can be integrated into it as particular vertices, this work consists mainly in the realization of an application displaying graphs. This leads us to investigate some techniques of graph drawing.

AVANT-PROPOS

Ce mémoire est le dernier pas de mon parcours estudiantin. Il marque ainsi la fin d'une période de ma vie qui m'a énormément apporté, aussi bien au niveau social qu'au niveau intellectuel. C'est pourquoi, par ces quelques mots, je voudrais témoigner de ma sincère reconnaissance aux personnes qui m'ont aidée tout au long de mon stage et de la rédaction de ce mémoire mais aussi au cours de mon passage aux Facultés Universitaires Notre-Dame de la Paix.

Je tiens à remercier :

l'Université de Brown, et tout particulièrement le Département Informatique, de m'avoir permis d'effectuer mon stage en son sein. Je les remercie de leur accueil et de leur patience ;

les Facultés Universitaires Notre-Dame de la Paix, dont le Département Informatique, de m'avoir permis de réaliser cette maîtrise en Informatique ;

le professeur P. Van Hentenrijck de m'avoir donné l'occasion de faire un stage aux États-Unis, dans une université telle que Brown ;

le professeur L. Michel de m'avoir guidée et conseillée dans la réalisation de l'utilitaire d'affichage de graphe. Je le remercie aussi pour sa patience et pour sa disponibilité ;

le professeur B. Le Charlier pour son soutien et ses encouragements. Je tiens également à le remercier pour sa disponibilité et son aide lors de la rédaction du présent mémoire ;

Grégory Dony pour les moments de travail, de détente, de sport, ... bref, pour la complicité qui nous a unis durant ce séjour aux États-Unis ;

les professeurs de la Faculté d'Informatique ;

tous les étudiants que j'ai rencontrés au cours de ces dernières années, tout particulièrement les membres des différents Cercles Info depuis 1998, ceux du Kot AGE et ceux du Nom de la Rose 2000, qui m'ont permis de m'épanouir aussi bien dans les études que dans les loisirs ;

les professeurs de l'Institut d'Enseignement Supérieur de Namur qui m'ont encouragée à aller plus loin ;

ma Maman, de n'avoir cessé de croire en moi et en la Vie malgré tout ... ;

Christiane et Jean-Pol, pour leur compréhension, leur soutien et leur confiance en moi ;

Lorette et Michel, Benjamin, Manu, Maryse, Florence et Yves, Mimie, Nicole, Marc et tous ceux qui m'ont soutenue de près ou de loin ;

Christophe, Cédric, Carmen et Xavier, Audrey et Adrian, Lionel, Éric, Emmeline, Amélie et Piou-Piou, Isabelle et Hugues, pour leur amitié ;

et, enfin, les étudiants de troisième maîtrise ainsi que Isabelle, Butch, Gregg, Cécile, Patrick et Karl.

TABLE DES MATIÈRES

PRÉFACE 7

CHAPITRE 1 - L'INTERPRÉTATION ABSTRAITE ET SON LIEN À LA THÉORIE DES GRAPHES 9

1.1 ANALYSE STATIQUE ET INTERPRÉTATION ABSTRAITE 10
1.1.1 L'analyse statique *10*
1.1.2 L'interprétation abstraite *10*
1.2 L'ENVIRONNEMENT ET LE STORE 12
1.3 LE PROJET JAVABINT 17

CHAPITRE 2 -LES ALGORITHMES D'AFFICHAGE D'UN GRAPHE REPRÉSENTANT L'ENVIRONNEMENT ET LE STORE D'UN PROGRAMME JAVA 19

2.1 RAPPELS DE LA THÉORIE DES GRAPHES 21
2.2 PARAMÈTRES DU DESSIN DE GRAPHE 26
2.2.1 Introduction *26*
2.2.2 Conventions *26*
2.2.3 Critères esthétiques *27*
2.2.4 Contraintes *27*
2.3 GRAPHE REPRÉSENTANT L'ENVIRONNEMENT ET LE STORE 28
2.4 LES ALGORITHMES UTILISÉS 31
2.4.1 Introduction *31*
2.4.2 Notation *31*
2.4.3 Le système de coordonnées *31*
2.4.4 Les structures de données liées au dessin du graphe *32*
2.4.5 Parcours du graphe *32*
2.4.6 Calculs préalables au dessin du graphe *34*
2.5 OPTIMISATIONS 69
2.5.1 Limiter la profondeur du dessin *69*
2.5.2 Zoom *71*
2.5.3 Autres propriétés *71*

CHAPITRE 3 - LES ASPECTS INTÉRESSANTS DE JAVA ET L'IMPLÉMENTATION 73

3.1 LE DESSIN EN JAVA 74
3.1.1 Le système de coordonnées *74*
3.1.2 Graphics *74*
3.1.3 Graphic2D *77*
3.2 LES ITÉRATEURS 79
3.3 LE PACKAGE FUNCTION 81
3.4 L'IMPLÉMENTATION 83
3.4.1 Les structures de données *83*
3.4.2 L'architecture *84*

CHAPITRE 4 - AMÉLIORATIONS À APPORTER 87

4.1 DESSIN DU GRAPHE 88
4.2 PERFORMANCE 88
4.3 PROPRIÉTÉS DE DESSIN 88
4.4 INTERFACE DE L'UTILITAIRE 89

CONCLUSION 91

BIBLIOGRAPHIE 93

ANNEXE 1 - CONSTRUCTION DU GRAPHE À DESSINER 95

ANNEXE 2 - STRUCTURES DE DONNÉES 99

PRÉFACE

Ce travail est le mémoire réalisé en vue de l'obtention du diplôme de Maître en Informatique. Il a été précédé d'un stage, qui s'est déroulé du 15 octobre 2000 au 31 janvier 2001 à l'Université de Brown, à Providence, aux États-Unis. Il fait suite au mémoire réalisé par Karl NOBEN en 2000 [NOBEN].

Ce stage avait pour but la réalisation d'un utilitaire d'affichage permettant la visualisation de l'état de la mémoire d'un programme Java, c'est-à-dire l'environnement et le store. Cet utilitaire a été développé en Java et dans le cadre de travaux de recherche visant à mettre au point "un analyseur statique générique de Java par interprétation abstraite".

Le chapitre 1er présente brièvement ce que sont l'analyse statique et l'interprétation abstraite, ainsi que le lien qui les unis à la théorie des graphes. Le projet JavAbInt est également présenté au sein de ce chapitre.

Dans le chapitre 2, les algorithmes utilisés pour l'affichage du graphe représentant un couple constitué d'un environnement et d'un store sont largement expliqués.

Le chapitre 3 insiste sur les aspects de la Programmation Orientée Objet et de Java qui semblent, de par l'importance qu'ils ont dans la réalisation de l'application, être dignes d'approfondissement.

Dans le chapitre 4, nous mettrons le doigt sur ce qui semble nécessiter des modifications et sur ce qui devrait être ajouté à l'utilitaire d'affichage.

Enfin, une conclusion donnera un aperçu de ce que nous a apporté ce mémoire.

CHAPITRE 1

L'INTERPRÉTATION ABSTRAITE ET SON LIEN À LA THÉORIE DES GRAPHES

L'utilitaire d'affichage de l'environnement et du store est réalisé dans le cadre d'un programme de recherche sur un analyseur statique générique de Java par interprétation abstraite.

Nous commencerons donc par présenter brièvement ce que nous entendons par analyse statique et par interprétation abstraite. Quelques mots concernant l'environnement et le store permettront de mieux comprendre ce en quoi le problème de l'affichage d'un graphe nous concerne. Enfin, nous présenterons brièvement le projet JavAbInt.

1.1 ANALYSE STATIQUE ET INTERPRÉTATION ABSTRAITE

Afin de mieux situer le contexte dans lequel a été développé l'utilitaire d'affichage de l'environnement et du store, quelques informations sur ce que sont l'analyse statique et l'interprétation abstraite semblent être utiles. Les explications qui suivent se réfèrent aux articles *[LECHARLIER]* et *[STEINDL]*.

1.1.1 L'analyse statique

L'analyse statique de programmes est le terme utilisé pour désigner l'ensemble des traitements qui peuvent être appliqués à un programme en dehors de son exécution proprement dite. *A contrario*, l'analyse dynamique permet de dériver des informations lors d'une exécution particulière du programme. Dans le cas de l'analyse dynamique, seule la valeur de l'expression au moment de l'exécution est considérée, et non la valeur de l'expression en générale.

C'est notamment au cours de l'implémentation, et principalement lors de la compilation, que l'analyse statique de programme est utilisée. Cette analyse porte ainsi sur le code source et permet d'obtenir des informations sur le programme. Celles-ci doivent être valides lors de toutes les exécutions possibles.

Les informations ainsi obtenues servent à optimiser le programme ou à vérifier que celui-ci est conforme à certains critères de correction. Par exemple, s'il s'agit d'optimiser le programme, l'analyse statique permet, dans le cas d'une boucle, de déterminer si la valeur d'une expression reste constante lors d'exécutions successives du corps de la boucle et, si c'est le cas, de déplacer cette expression à l'extérieur de cette boucle. Dans le cas de la vérification, elle permet de vérifier la conformité des expressions du programme à la syntaxe du langage ou de s'assurer de la correction de propriétés sémantiques telles que le caractère bien typé d'expressions complexes.

Cependant, pendent longtemps, les diverses techniques d'analyses statiques utilisées dans les compilateurs furent basées sur des raisonnements empiriques plutôt que sur des théories générales. Or la correction de ces techniques est cruciale puisque le programme, avant d'être soumis à une quelconque optimisation, doit être correct. Mais la preuve de correction d'une méthode d'analyse statique est une tâche très difficile étant donné la complexité des langages de programmation à analyser. C'est dans ce cadre qu'intervient l'interprétation abstraite.

1.1.2 L'interprétation abstraite

Introduit par P. et R. Cousot, le concept d'interprétation abstraite désigne un cadre mathématique général pour l'analyse statique. Elle permet d'exprimer les méthodes d'analyse statique existantes et de prouver leur correction.

Le principe de base de l'interprétation abstraite est d'exécuter le programme à analyser sur un domaine non standard, au lieu du domaine de calcul normal. Le domaine de l'analyse statique est qualifié de domaine abstrait par opposition au domaine concret utilisé en dehors de cette analyse.

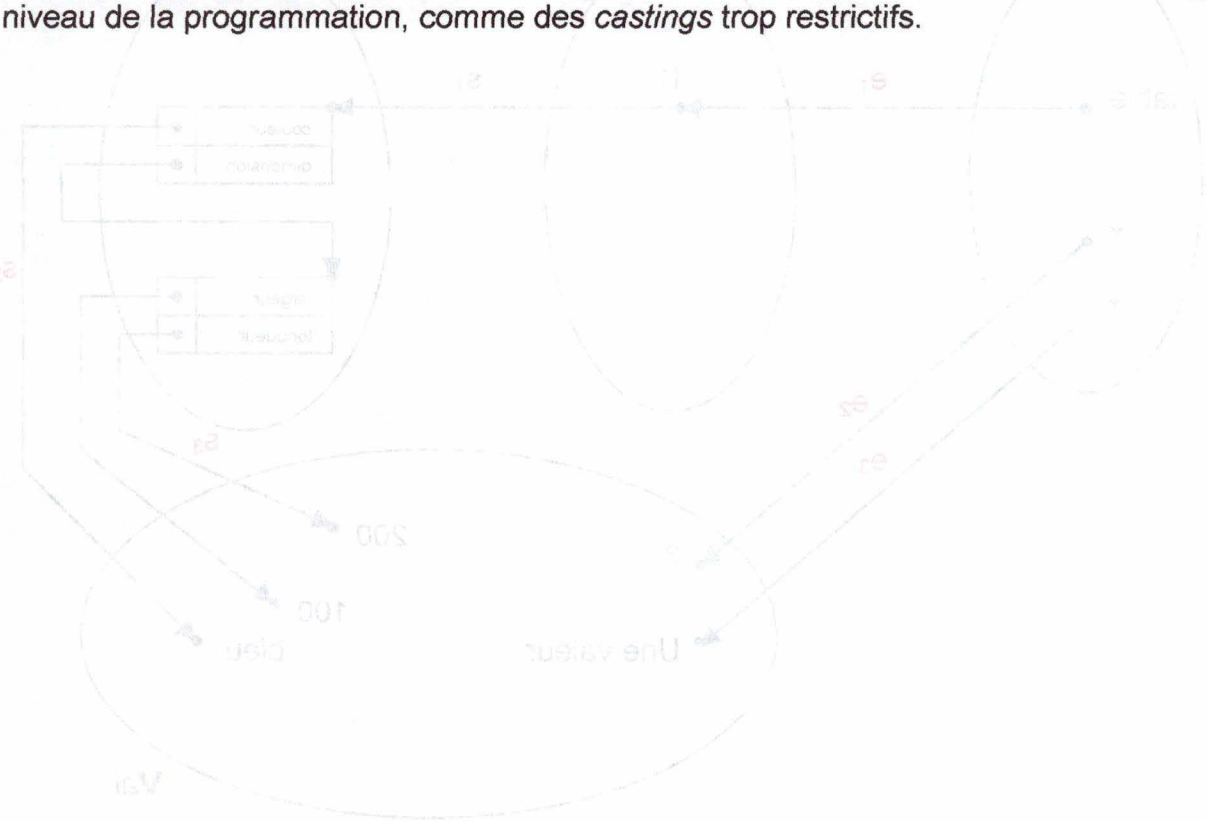
Il s'agit donc de déterminer un domaine abstrait sur lequel l'analyse statique pourra être effectuée. Ce domaine abstrait n'est autre qu'un ensemble d'éléments dont les valeurs concrètes présentent des caractéristiques communes. Ces éléments doivent ainsi représenter des propriétés utiles du domaine standard.

De par la définition du domaine abstrait, un élément de celui-ci pouvant représenter plusieurs éléments du domaine concret, l'analyse statique par interprétation abstraite peut entraîner une perte de précision. Cependant, cette perte peut être compensée par le choix d'un domaine abstrait adéquat. De plus, les informations résultant de l'analyse statique sont obtenues une fois pour toute, elles ne dépendent pas d'une exécution particulière, par opposition aux informations découlant d'une analyse dynamique.

Afin que les méthodes d'interprétation abstraite soient utilisables en pratique, les calculs sur le domaine abstrait doivent être réalisés de manière suffisamment efficace et en un temps fini.

L'interprétation abstraite a commencé par être utilisée en programmation impérative classique. Mais c'est avec l'avènement de la programmation déclarative, dont la programmation logique, que l'interprétation abstraite s'est réellement fait reconnaître. En effet, les langages déclaratifs donnent beaucoup moins d'informations quant à la manière dont les résultats sont obtenus et fournissent ainsi plus d'opportunités d'analyse et donc d'optimisation.

Le paradigme qui nous intéresse particulièrement est celui de la programmation orientée objet. Les mécanismes qui la caractérisent, tel que l'héritage, le polymorphisme ou les liens dynamiques (*dynamic binding*, c'est-à-dire que la méthode réellement exécutée lors d'un appel est déterminée à l'exécution), semblent être d'un intérêt particulier dans le cadre de l'interprétation abstraite. En effet, ces mécanismes entraînent la multiplicité des redéfinitions de classes ou de méthodes. L'analyse statique permet, dans ce cas, le remplacement de liens dynamiques par des liens statiques plus efficaces ainsi que la détection d'erreurs d'incohérences au niveau de la programmation, comme des *castings* trop restrictifs.



1.2 L'ENVIRONNEMENT ET LE STORE

Lors de l'exécution d'un programme, nous représentons l'état courant de la mémoire du système par plusieurs éléments dont l'environnement et le store. La finalité de l'utilitaire d'affichage est de permettre à l'utilisateur de visualiser ces deux éléments.

En Java, toutes les valeurs des objets sont des références, sauf dans le cas des types scalaires tels que les entiers, les chaînes de caractère ... La définition de l'environnement et celle du store sont donc sensiblement différentes de celles habituellement présentées.

L'**environnement** est une fonction qui permet d'associer à chaque nom de variable une référence s'il s'agit d'un objet ou une valeur s'il s'agit d'un type scalaire. Cette fonction associe donc un élément de l'ensemble des noms de variable INomVar à un élément de l'ensemble des références ou locations, ILoc, ou à un élément de l'ensemble des valeurs **V**al.

Le **store** est une fonction qui associe une location au contenu de l'objet qu'elle référence. Cette fonction associe donc un élément de l'ensemble des locations ILoc à un élément de l'ensemble des objets **O**bjet. Chaque élément de ce dernier regroupe plusieurs locations qui référencent chacune la valeur associée à un des attributs de l'objet.

Exemple 1.1

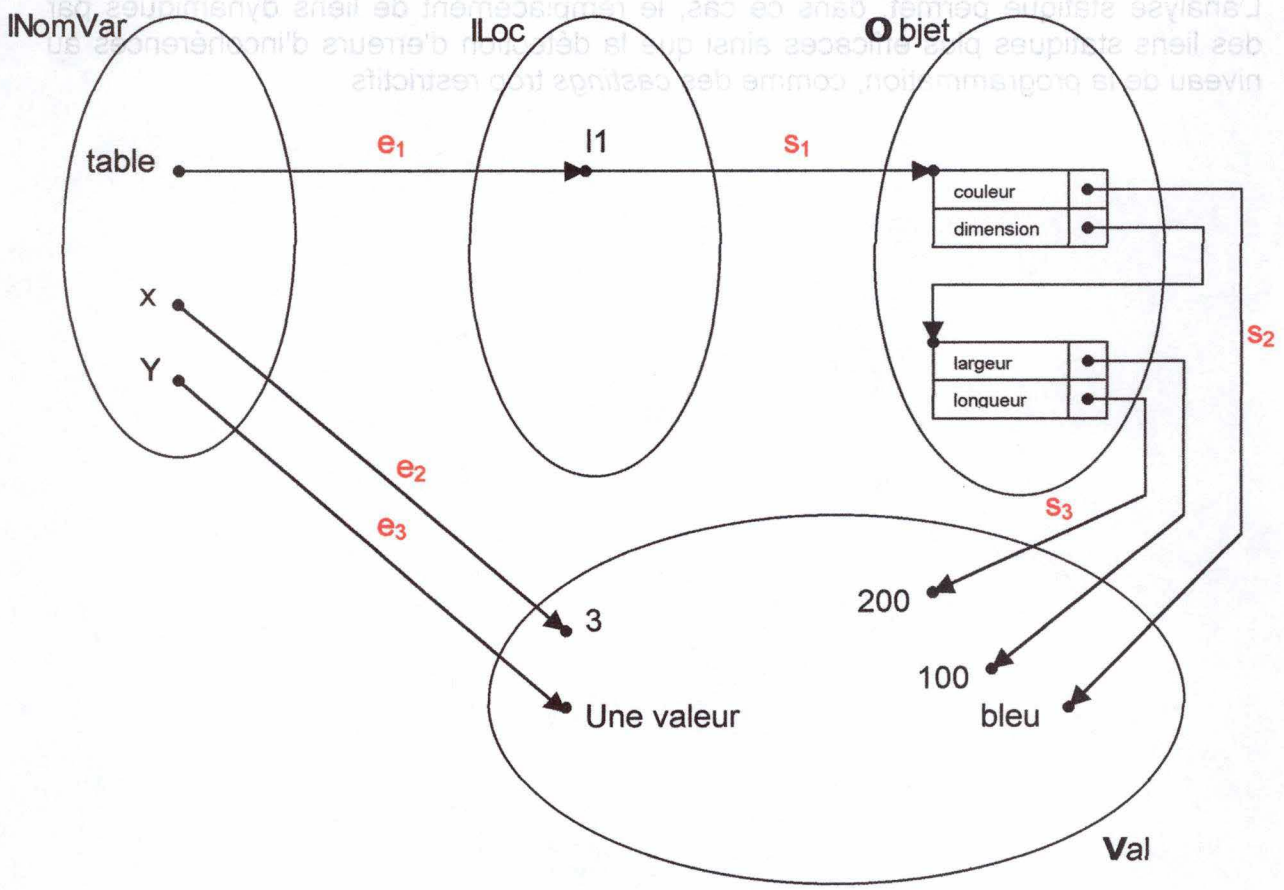


Figure 1.1 : L'environnement et le store.

A chaque évaluation d'une instruction du programme, l'environnement et le store sont susceptibles de changer. Ci-après se trouve un exemple permettant de visualiser ce qui se passe concernant l'environnement et le store au cours de l'exécution d'un programme.

Exemple 1.2

- *Programme Java*

```
public class Exemple {

        // Attributs
        int data1;
        int data2;
        int resultat;
        Addition addition;

        // Constructeur
        public Exemple (int donnee1, int donnee2) {                    (α)
                data1 = donnee1;
                        data2 = donnee2;
                addition = new Addition(data1, data2);                 (β)
                resultat = addition.getResultat();
                System.out.println(data1 + " + " + data2 + " = " + resultat);
        }

        // main
        public static void main(String[] args) {
                Exemple exemple = new Exemple(5, 9);
        }
}

public class Addition {
        int resultat;

        // Constructeur
        public Addition (int donnee1, int donnee2) {
                resultat = donnee1 + donnee2;
        }

        // Méthode
        public int getResultat() {
                return resultat;
        }
}
```

- *Environnement et Store*

  Nous allons observer deux des états possibles du système pour le programme *Exemple*.

  Juste avant l'exécution de la ligne *(α)*, la méthode *main* du programme a fait appel au constructeur de la classe *Exemple*. Les éléments de l'environnement sont donc les suivants :

Nom de variable	*Location*
this	l1
donnee1	5
donnee2	9

Tableau 1.1

Le store se compose des éléments suivants :

Location	*Nom du champ*	*Type*	*Valeur*
l1	data1	int	null
	data2	int	null
	resultat	int	null
	addition	Addition	null

Tableau 1.2

Après l'exécution de la ligne *(β)*, l'environnement et le store deviennent :

Nom de variable	*Location*
this	l1
donnee1	5
donnee2	9

Tableau 1.3

Location	*Nom du champ*	*Type*	*Valeur*
l1	data1	int	5
	data2	int	9
	resultat	int	14
	addition	Addition	l4
l4	resultat	int	14

Tableau 1.4

La valeur correspondant à un nom de variable peut être simple, c'est-à-dire un entier, un booléen ou une chaîne de caractères, ou elle peut être une référence à un objet, ou encore elle peut être indéterminée. Dans le cas où il s'agirait de la référence à un objet, la valeur de l'objet est l'ensemble des attributs de la classe correspondant à cet objet. Chacun de ces attributs est en fait une référence vers un objet ou une valeur simple, et ainsi de suite. La structure du store est donc semblable à celle d'un graphe.

L'exemple suivant permet de visualiser le fait que le store peut être représenté par un graphe.

Exemple 1.3

Prenons une classe *Table* définie comme suit :

```
class Table {
        // Attributs
        int nombrePieds;
        int hauteur;
        Dimension dimension;
        String couleur;
        // Méthodes ...
        public Table() {
                nombrePieds = 4;
                hauteur = 80;
                dimension = new dimension( 150, 200);
                couleur = "bleu";
        }
        // main
        public static void main(String[] args) {
                Exemple table = new Table();
        }
}
class Dimension {
        // Attributs
        int longueur;
        int largeur
        // Méthodes ...
        public Dimension (int larg, int long) {
                longueur = long;
                largeur = larg;
        }
}
```

Le graphe correspondant au store obtenu juste après l'appel de la méthode *main* est le suivant :

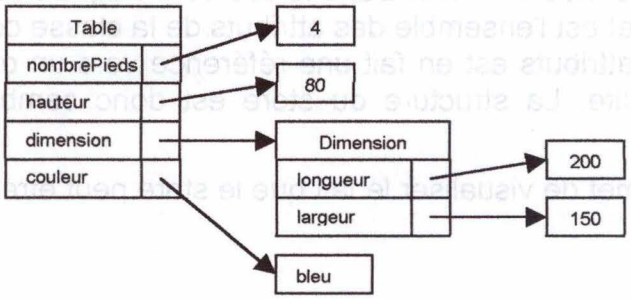


Figure 1.2 : Graphe représentant le store.

Le graphe représentant le store peut être complété par les éléments de l'environnement. Ces éléments seront en effet des sommets particuliers du graphe. Dans ce cas, le store est simplement composé de *this*, la référence de l'objet du type Table.

1.3 LE PROJET JAVABINT

Le projet JavAbInt a pour objectif d'appliquer les techniques de l'interprétation abstraite au langage de programmation orienté objet Java.

Ce projet a été initié par le professeur B. Le Charlier des Facultés Universitaire Notre-Dame de la Paix de Namur et par mademoiselle I. Pollet, chercheuse à la faculté d'informatique de la même université. L'équipe se compose également des professeurs A. Cortesi de l'Université Ca'foscari de Venise et P. Van Hentenrijck de l'Université de Brown à Providence. Chaque année, des étudiants se joignent à cette équipe dans le cadre de leur mémoire. Une thèse de D.E.A. a également été réalisée dans le cadre de ce projet par I. Pollet *[POLLET]*.

Durant l'année 2000, plusieurs étudiants ont travaillé en collaboration avec ces professeurs dans le cadre de leur mémoire. C. Hayez et P. Hendricks avaient pour objectif la réalisation d'un *parser* et d'un *type checker* pour un sous-langage de Java, et K. Noben s'est intéressé à l'implémentation d'un utilitaire d'affichage de l'environnement et du store d'un programme Java.

Cette année, deux étudiants se sont joints à l'équipe, chacun ayant pour tâche de continuer ce qui a déjà été réalisé. G. Dony a poursuivit la réalisation du *parser* et du *type checker*; quant à l'utilitaire d'affichage, c'est le sujet du présent mémoire.

CHAPITRE 2

LES ALGORITHMES D'AFFICHAGE D'UN GRAPHE REPRÉSENTANT L'ENVIRONNEMENT ET LE STORE D'UN PROGRAMME JAVA

Les graphes sont souvent utilisés lorsqu'il s'agit de représenter un ensemble d'éléments et de liens entre ces éléments. Comme nous l'avons vu précédemment, l'environnement et le store peuvent être représentés par un graphe. Le concept de graphe n'est pas nouveau. Il était déjà utilisé par Euler au XIII^{ème} siècle. De nombreux travaux ont été réalisés depuis, et tout cela constitue ce que nous appelons aujourd'hui la théorie des graphes. Avant d'expliquer la démarche qui a été suivie pour élaborer le programme d'affichage d'un graphe, quelques rappels concernant cette théorie s'imposent.

Après cela, nous proposerons un aperçu théorique des paramètres importants du dessin de graphe. Trois nouveaux concepts sont introduits pour exprimer les paramètres du dessin de graphe : les conventions, les critères esthétiques et les contraintes.

Ensuite, une description plus précise du problème dont nous nous occupons sera réalisée. Les paramètres de dessin propres à notre cas y seront précisés ainsi que quelques éléments nécessaires à la bonne compréhension de la suite des explications. Nous verrons notamment que, pour chaque élément de l'environnement, un arbre est extrait du graphe à l'aide d'un algorithme de parcours en profondeur d'abord.

Les éléments théoriques présentés et le problème spécifié, nous présenterons enfin les algorithmes qui ont été utilisés pour afficher un graphe dans le cadre de l'utilitaire d'affichage de l'environnement et du store. Il ne s'agit donc pas d'algorithmes permettant l'affichage de n'importe quel graphe mais bien d'un graphe représentant l'environnement et le store d'un programme Java.

L'algorithme de parcours en profondeur d'abord est l'algorithme de base de notre utilitaire. Il est donc le premier à être abordé.

Se basant sur cet algorithme de parcours de graphe, nous présenterons les algorithmes calculant divers éléments nécessaires au dessin du graphe. Ces calculs sont regroupés en sept modules. Le dernier algorithme est celui qui réalise le dessin en lui-même.

En guise de conclusion, nous aborderons les divers éléments qui ont été ajoutés au programme afin de raffiner le dessin tels que la limitation de la profondeur du dessin et les zooms.

2.1 RAPPELS DE LA THÉORIE DES GRAPHES

L'ensemble des définitions ci-après permet au lecteur de mieux comprendre les termes liés à la théorie des graphes qui seront utilisés par la suite. Pour plus de détails, nous proposons au lecteur de se référer au syllabus du cours sur la théorie des graphes donné par le professeur J. Fichefet [FICHEFET].

graphe Un graphe est un ensemble fini ou dénombrable de sommets (ou nœuds) reliés par des arcs (graphe orienté) ou des arêtes (graphe non orienté). Le graphe **G** est donc un couple (**X**,**U**) constitué d'un ensemble fini **X** de sommets et d'un ensemble **U** d'arcs.

Les éléments de **U** sont dénotés par une paire de sommets (a,b) avec a, b $\in$ **X**.

Graphiquement, les sommets sont représentés par des points et les arêtes par des lignes reliant ces points.

graphe orienté Un graphe orienté est un graphe dans lequel une direction est associée à chaque arc. Ceci permet de distinguer les deux extrémités de l'arc, l'une étant l'extrémité initiale et l'autre l'extrémité terminale.

Graphiquement, la direction de l'arc est représentée par une flèche orientée vers l'extrémité terminale.

Exemples : avec **X** = $\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}$

U = $\{u_1,u_2,u_3,u_4,u_5,u_6,u_7,u_8,u_9\}$

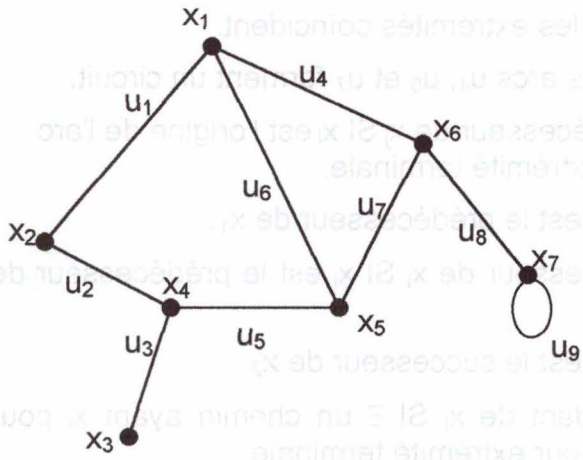


Figure 2.1 : Graphe.

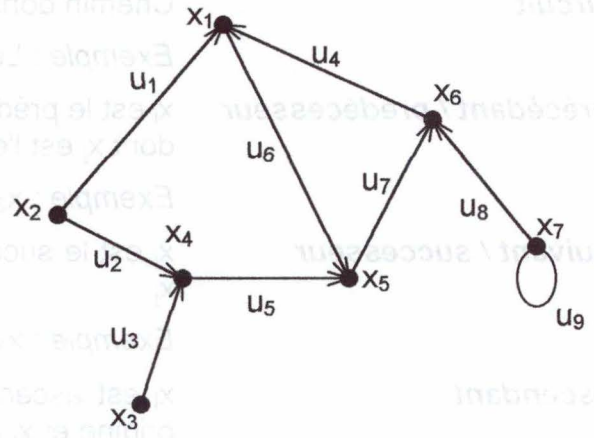


Figure 2.2 : Graphe orienté.

extrémité initiale / origine Soit un arc u = (x_i,x_j), x_i est l'origine de l'arc.

Exemple : Le sommet x_2 est l'origine de l'arc u_1.

extrémité terminale / finale Soit un arc u = (x_i,x_j), x_j est l'extrémité terminale de l'arc.

Exemple : Le sommet x_1 est l'extrémité terminale de l'arc u_1.

sommets adjacents

Deux sommets sont adjacents SI ils sont distincts *et* SI $\exists$ un arc $u = (x_i,x_j)$ ou $v = (x_j,x_i)$.

Exemple : Les sommets x_1 et x_2 sont adjacents.

boucle

Une boucle est un arc dont l'extrémité initiale et l'extrémité finale sont identiques.

Noté $u = (x_i,x_i)$.

Exemple : $u_9 = (x_7,x_7)$

arc incident

Un arc est incident à un nœud si ce nœud est une extrémité de l'arc.

Exemple : L'arc u1 est incident à x_1 et x_2.

degré d'un sommet

Nombre d'arcs qui ont ce sommet comme extrémité (sans tenir compte des boucles).

Exemple : Le degré de x_1 est 3.

chemin

Suite d'arcs telle que l'extrémité terminale de u_i est l'origine de u_{i+1}.

Exemple : Les arcs u_2, u_5, u_7 et u_4 forment un chemin μ.

longueur du chemin

Nombre d'arcs appartenant au chemin.

Exemple : La longueur du chemin μ est 4.

sommet accessible

x_i est accessible à partir de x_j SI $\exists$ un chemin de x_j à x_i.

Exemple : Le sommet x_1 est accessible à partir de x_3.

circuit

Chemin dont les extrémités coïncident.

Exemple : Les arcs u_4, u_6 et u_7 forment un circuit.

précédant / prédécesseur

x_i est le prédécesseur de x_j SI x_i est l'origine de l'arc dont x_j est l'extrémité terminale.

Exemple : x_2 est le prédécesseur de x_1.

suivant / successeur

x_j est le successeur de x_i SI x_i est le prédécesseur de x_j.

Exemple : x_1 est le successeur de x_2.

ascendant

x_i est ascendant de x_j SI $\exists$ un chemin ayant x_i pour origine et x_j pour extrémité terminale.

Exemple : $\{x_1, x_2, x_3, x_4, x_5, x_7\}$ sont les ascendants de x_6.

descendant

x_j est descendant de x_i SI x_i est ascendant de x_j.

Exemple : $\{x_1, x_4, x_5, x_6\}$ sont les descendants de x_2.

Pour définir la notion d'arbre, il est nécessaire d'introduire certaines notions liées aux graphes non orientés. En effet, la définition d'arbre ne tient pas compte de l'orientation éventuelle des arcs. Les exemples concernant les arbres feront donc référence à la figure 2.1.

pseudo-chaîne

Une pseudo-chaîne est une suite d'arêtes telle que chaque arête u_i est reliée à u_{i-1} par une extrémité et à u_{i+1} par l'autre extrémité.

Exemple : $\{u_1, u_2, u_3\}$

chaîne

Une chaîne est une pseudo-chaîne dont les extrémités coïncident.

Exemple : $\{u_1, u_2, u_5, u_7, u_4\}$

relation de connexité simple

La relation de connexité simple, R_f, est une relation d'équivalence (c'est-à-dire réflexive, symétrique et transitive) sur X telle que $\forall x_i, x_j \in X : x_i R_f x_j \Leftrightarrow x_i = x_j$ ou $\exists$ une chaîne entre x_i et x_j.

composante simplement connexe

Une composante simplement connexe est une classe d'équivalence produite sur X par R_f. C'est-à-dire X', une partition de X, telle que $\forall x_i \in X' : x_i \in$ chaîne dans R_f.

graphe simplement connexe

Un graphe est simplement connexe SSI

$\forall x_i, x_j \in X$: xi et xj sont joints par une chaîne.

Exemple : Le graphe est simplement connexe.

arbre

Un arbre est un graphe simplement connexe sans cycle.

Exemple :

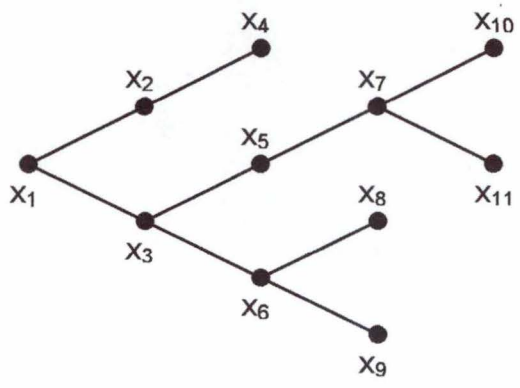


Figure 2.3 : Arbre.

Dans notre cas, nous aurons à faire à des graphes orientés. Il est donc nécessaire d'associer un sens à chacun des arcs de l'arbre.

Exemple :

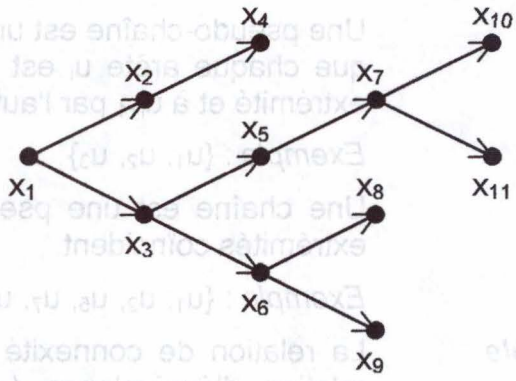


Figure 2.4 : Arbre orienté.

Cette orientation nous permet entre autre de déterminer la racine et les feuilles de l'arbre.

racine Sommet ascendant de tous les sommets de l'arbre.

Exemple : x_1 est la racine du graphe.

écart Longueur du plus court chemin entre deux sommets.

Exemple : L'écart entre x_1 et x_4 est de 2.

feuille Sommet qui n'est l'origine d'aucun arc.

Exemple : x_4, x_8, x_9, x_{10} et x_{11} sont les feuilles de l'arbre.

S'il existe plusieurs composantes simplement connexes, et donc plusieurs racines, on parlera d'une ***forêt***.

graphe partiel

Soit G = (X,U) : G'(X,V) est un graphe partiel de G SI V est un sous-ensemble de U.

Exemple : Graphe partiel du graphe de la figure 2.2.

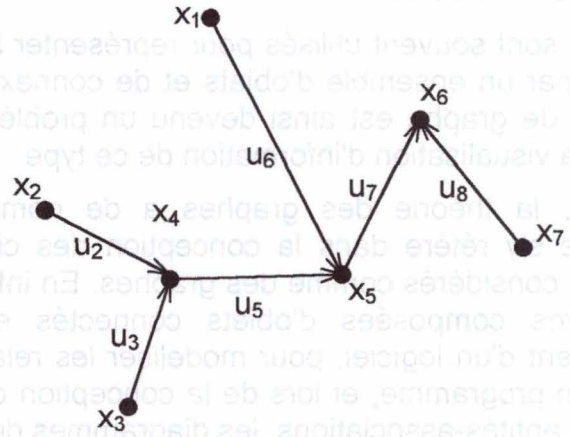


Figure 2.5 : Graphe partiel.

sous-graphe

Soit G = (X,U) : G'(Y,V) est un sous-graphe de G SI V est un sous-ensemble de U *et* SI Y est un sous-ensemble de X.

Exemple : Sous-graphe du graphe de la figure 2.2.

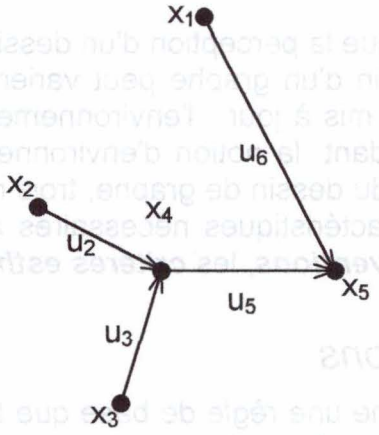


Figure 2.6 : Sous-graphe.

graphes isomorphes

Les graphes G = (X,U) et H = (Y,V) sont isomorphes SI $(x_i,x_j) \in U \Leftrightarrow (y_i,y_j) \in V$

2.2 PARAMÈTRES DU DESSIN DE GRAPHE

2.2.1 Introduction

Les graphes sont souvent utilisés pour représenter les informations qui peuvent être modélisées par un ensemble d'objets et de connexions entre ces objets. Le dessin automatique de graphe est ainsi devenu un problème crucial pour les applications permettant la visualisation d'information de ce type.

En pratique, la théorie des graphes a de nombreuses applications. En effet, l'électronique s'y réfère dans la conception des circuits intégrés et imprimés, qui peuvent être considérés comme des graphes. En informatique, on rencontre souvent des structures composées d'objets connectés entre eux, notamment lors du développement d'un logiciel, pour modéliser les relations de dépendance entre les modules d'un programme, et lors de la conception d'un système d'information, dans les schémas entités-associations, les diagrammes de flux ...

Le problème du dessin de graphe quelconque est un problème complexe sur lequel portent de nombreux travaux de recherche. Ceux-ci ont notamment permis à G. Di Battista *et al.* [DBETT] de mettre en évidence divers paramètres du dessin de graphe.

Un paramètre important, dans l'élaboration d'un algorithme de dessin de graphe, est de savoir de quel type de graphe il s'agit (cyclique, dirigé, planaire ...). En effet, certains algorithmes de dessin sont spécifiques ou mieux appropriés à un type particulier.

De plus, étant donné que la perception d'un dessin peut être différente d'un individu à l'autre, et que le dessin d'un graphe peut varier selon le domaine d'application, un autre paramètre a été mis à jour : l'environnement dans lequel le dessin du graphe va être utilisé. Cependant, la notion d'environnement étant trop abstraite pour être utilisée dans le cadre du dessin de graphe, trois nouveaux concepts ont été dégagés pour exprimer les caractéristiques nécessaires à un dessin de graphe lisible. Ces concepts sont les ***conventions***, les ***critères esthétiques*** et les ***contraintes***.

2.2.2 Conventions

Une convention exprime une règle de base que le dessin doit absolument satisfaire. Les conventions peuvent être très complexes et porter sur des détails du dessin. Il s'agit souvent de règles portant sur la représentation des sommets et des arcs.

Voici une liste de conventions souvent prises en compte :

Dessin "Polyligne"	Chaque arc est représenté par une ligne brisée.
Dessin "Ligne droite"	Chaque arc est représenté par une ligne droite.
Dessin "Orthogonale"	Chaque arc est représenté comme une suite de segments horizontaux et verticaux.
Dessin "Grille"	Les nœuds, les croisements d'arcs et les points de routages des arcs sont alignés sur une grille.
Dessin "Planaire"	Deux arcs ne peuvent se croiser.

Exemple 2.1

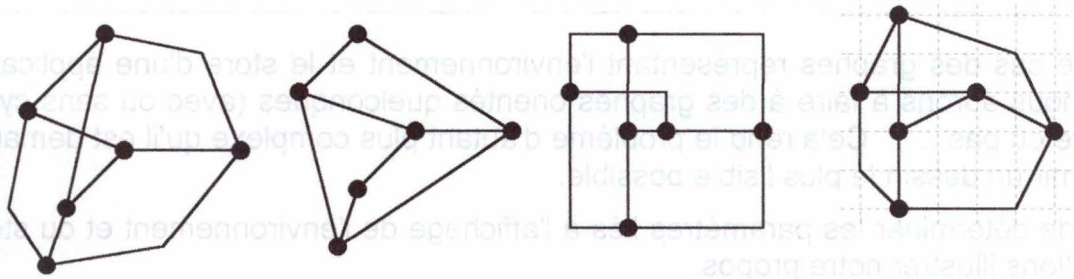


Figure 2.7 : Conventions de dessin de graphe.

Les dessins "Ligne droite" et "Orthogonale" sont des cas particuliers du dessin "Polyligne". Chacune de ces conventions a des avantages et des inconvénients, et certaines ne conviennent pas à certains types de graphes. Tous les graphes ne sont, par exemple, pas planaires et ne permettent donc pas de respecter une éventuelle convention de planarité.

2.2.3 *Critères esthétiques*

Les critères esthétiques spécifient les propriétés graphiques qu'il convient de respecter, dans la mesure du possible, pour augmenter la lisibilité du dessin. Parmi les critères esthétiques, on retrouve parfois une convention qu'il n'est pas possible de satisfaire dans tous les cas. La planarité est un exemple d'une telle situation dans le sens où elle est souhaitable mais pas réalisable pour tous les graphes.

Les critères esthétiques habituels sont la minimisation des croisements entre arcs, la minimisation de la surface de dessin, la minimisation de la somme des longueurs des arcs, la minimisation du nombre total de points de routage le long des arcs, la mise en évidence des symétries du graphe, la maximisation de l'angle le plus petit entre deux arcs incidents au même nœud ...

2.2.4 *Contraintes*

Les contraintes sont des règles qui ne concernent pas la totalité du graphe mais un sous-graphe et son dessin.

On retrouve, parmi les critères de dessin, des contraintes telles que placer un nœud donné au centre du dessin, placer un nœud donné à la limite extérieure du dessin, dessiner un chemin de manière horizontale en allant de la gauche vers la droite ...

2.3 GRAPHE REPRÉSENTANT L'ENVIRONNEMENT ET LE STORE

Dans le cas des graphes représentant l'environnement et le store d'une application Java, nous aurons à faire à des graphes orientés quelconques (avec ou sans cycle, planaire ou pas ...). Cela rend le problème d'autant plus complexe qu'il est demandé de fournir un dessin le plus lisible possible.

Avant de déterminer les paramètres liés à l'affichage de l'environnement et du store, nous allons illustrer notre propos.

Exemple 2.2

Cet exemple permet de visualiser la représentation graphique des environnements et stores décrits à l'exemple 1.2.

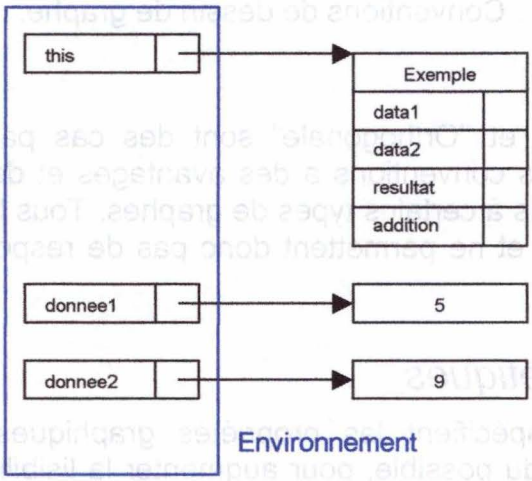


Figure 2.8 : Environnement du tableau 1.1 et store du tableau 1.2.

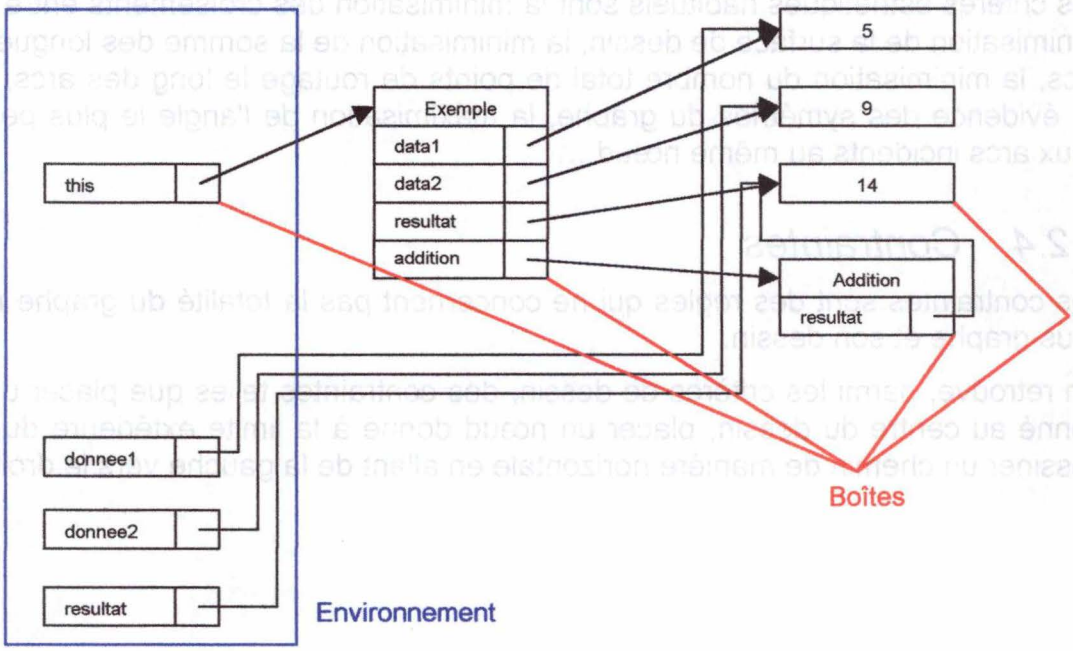


Figure 2.9 : Environnement du tableau 1.3 et store du tableau 1.4.

Ces dessins nous permettent de mettre l'accent sur certains aspects particuliers du problème de l'affichage de l'environnement et du store.

En effet, la surface occupée par les nœuds ne se limite pas à un point mais recouvre un rectangle de plus ou moins grande taille. La représentation graphique d'un nœud, qui sera dorénavant appelé ***boîte***, est celle d'un élément de l'environnement ou du store. Or il s'agit d'objet Java, c'est-à-dire qu'il se peut qu'un objet soit l'instance d'une classe comportant plusieurs attributs. Une boîte se compose donc au minimum d'un compartiment, contenant, dans le cas de l'environnement, le nom de la variable représentée et, dans le cas du store, le type de l'objet référencé par la variable ou sa valeur. En plus de cela, la boîte contient autant de compartiments qu'il y a d'attributs. La hauteur de cette boîte est donc variable, mais nous avons d'ores et déjà décidé que la largeur de celle-ci serait la même pour tous les nœuds.

Un autre facteur également important est le fait que l'origine d'un arc se situe du coté droit du rectangle représentant le nœud, à hauteur de l'attribut auquel il correspond, alors que le point correspondant aux extrémités terminales des arcs ayant le nœud pour destination se trouve du coté gauche.

Dans le but de rendre l'affichage du graphe le plus clair possible, nous avons déterminé les conventions, les facteurs esthétiques et les contraintes suivants.

Les ***conventions*** sont au nombre de trois :

* aucun arc ne peut traverser un nœud ;
* les arcs ne peuvent pas se recouvrir ;
* les nœuds sont disposés selon une structure d'arbre définie par un parcours en profondeur d'abord.

Étant donné que les nœuds sont représentés par des rectangles, les conventions justifient le fait que les nœuds sont alignés sur une grille et que les arcs sont dessinés parallèlement aux côtés des rectangles représentant les nœuds. Les points de routage des arcs sont donc également alignés sur la grille.

Exemple 2.3

Cet exemple illustre les trois conventions présentées ci-dessus.

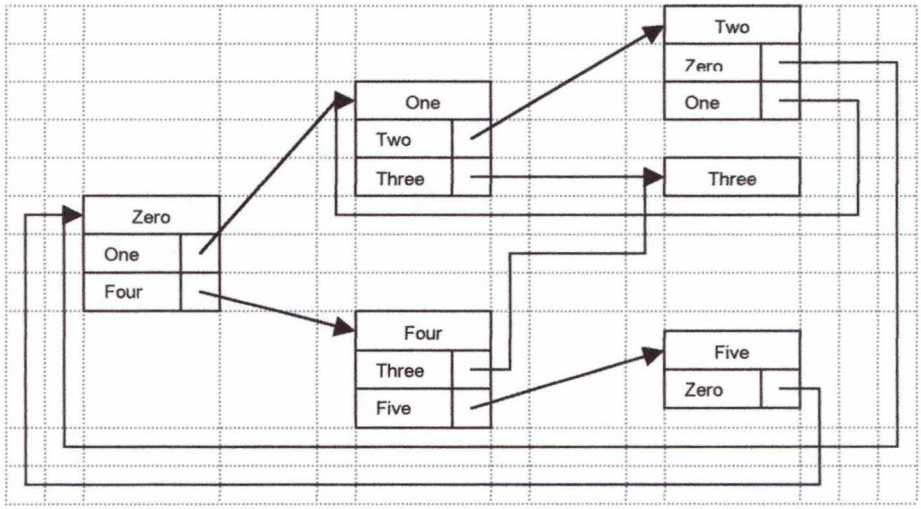


Figure 2.10 : Graphe quelconque avec la grille d'alignement.

Un ***critère esthétique*** est ajouté, et consiste en la minimisation des croisements entre arcs.

L'unique ***contrainte*** est que les nœuds faisant partie de l'environnement soient chacun la racine d'un arbre et soient rassemblés sur le côté gauche du dessin.

L'utilitaire d'affichage de graphe sera élaboré en accord avec ces paramètres.

2.4 LES ALGORITHMES UTILISÉS

2.4.1 Introduction

Pour commencer, nous allons expliquer les notations qui seront utilisées pour la description des algorithmes ainsi que le système de coordonnées qui est à la base des calculs effectués.

Ensuite, nous énumérerons les différentes structures de données utilisées en les décrivant très brièvement, de plus amples explications à leur propos étant disponibles dans l'annexe 2.

Après cela, nous présenterons l'algorithme de parcours du graphe et ensuite les différents calculs préalables aux dessins à proprement dit.

Le dessin du graphe sera l'étape finale de cette section.

2.4.2 Notation

Pour la description d'un **algorithme**, la notation suivante sera dorénavant utilisée :

Données :	Paragraphe contenant les paramètres de l'algorithme.
Résultat :	Paragraphe contenant le résultat de l'algorithme.
Utilise :	Paragraphe contenant les éléments utilisés par l'algorithme ne faisant pas partie des structures de données décrites.
Algorithme :	Paragraphe contenant l'algorithme en lui-même.

2.4.3 Le système de coordonnées

Avant de présenter les algorithmes qui permettent le dessin d'un graphe, il est important de donner quelques informations quant à la manière dont nous allons dessiner chaque élément du graphe.

Nous considérons que l'utilitaire d'affichage dessine le graphe sur un plan comme s'il s'agissait d'une feuille de papier. Ce plan est défini par les axes X et Y du système de coordonnées.

Exemple 2.4

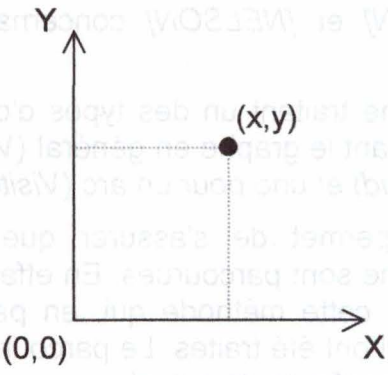


Figure 2.11 : Système de coordonnées du plan de dessin.

Chaque élément est donc dessiné en fonction de ce système de coordonnées. En effet, pour dessiner une droite, on a besoin de sa coordonnée d'origine et de celle de destination, et pour dessiner un rectangle, on a besoin de la coordonnée de son coin supérieur gauche ainsi que de sa hauteur et de sa largeur ...

Dans le cadre de cet utilitaire, le plan de dessin a pour unité de mesure le pixel.

2.4.4 Les structures de données liées au dessin du graphe

La représentation du graphe nécessite l'utilisation de trois types d'objets (ou classes) représentant un graphe, un nœud et un arc. Chacune de ces classes contient des attributs, c'est-à-dire des informations particulières la concernant, et des méthodes permettant d'interagir avec les objets de la classe en question. Par souci de clarté, nous présentons ci-après les notations utilisées pour désigner les attributs et les méthodes, ainsi que celles liées aux séquences.

Pour désigner l'attribut d'un objet, nous utiliserons la notation suivante :

attribut(objet) où *attribut* représente le nom de l'attribut référencé
et *objet* représente le nom de l'objet dont nous désirons la valeur de l'attribut.

Un appel de méthode pouvant renvoyer ou non un résultat, nous proposons deux formes d'appel de méthode :

- *Procédure NomMéthode(paramètre1, ... , paramètreN)*
  lorsque la méthode ne renvoie aucun résultat ;
- *Fonction résultat NomMéthode(paramètre1, ... , paramètreN)*
  lorsque la méthode renvoie un résultat.

En plus de cela, nous utilisons des séquences. Nous considérons qu'une séquence est un ensemble fini dont chaque élément est associé à un indice, c'est-à-dire un nombre entier compris entre 0 et N-1, N étant le nombre total d'éléments de la séquence. Ces éléments sont également triés en ordre croissant sur cet indice. Le $i^{ème}$ élément d'une séquence est identifié par *nomSéquence[i]*.

2.4.5 Parcours du graphe

Pour parcourir le graphe, on utilise un algorithme (ALGO1) de recherche en profondeur d'abord, communément appelé DFS pour *Depth First Search*. Il s'agit d'un algorithme récursif, ce qui veut dire que le traitement d'un nœud est terminé lorsque tous ses descendants ont été traités. Pour définir cet algorithme, nous avons utilisé les articles *[EPPSTEIN]* et *[NELSON]* concernant la recherche dans un graphe.

Il y a trois procédures, chacune traitant un des types d'objet composant le graphe, c'est-à-dire une procédure traitant le graphe en général (*VisitGraphe*), une traitant un nœud en particulier (*VisiteNœud*) et une pour un arc (*VisiteArc*).

La procédure *VisiteGraphe* permet de s'assurer que toutes les composantes simplement connexes du graphe sont parcourues. En effet, lorsque le parcours d'une composante est achevé, c'est cette méthode qui, en parcourant la séquence des nœuds, vérifie que tous ceux-ci ont été traités. Le parcours de la séquence se fait en suivant l'ordre croissant des identifiants de nœud.

Les procédures *VisiteNœud* et *VisiteArc* rendent toute opération sur les nœuds ou les arcs possibles.

ALGO1 – Parcours du graphe

Données : Un graphe.

Résultat : /

Utilise : Visite[] – une séquence de booléens permettant de savoir si un nœud doit encore être visité ou pas. Elle contient autant d'éléments que de nœuds dans le graphe. Chacun de ces éléments est initialisé à VRAI.

Nœuds[] – une séquence comprenant tous les nœuds du graphe.

Suivants[] – une séquence associée à un nœud et qui contient les arcs menant aux suivants de ce nœud.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    Nœuds ← nœuds(graphe)
    i ← 0
    Tant que i < taille(Nœuds)
    Début
        VisiteNœud(Nœuds[i])
        i ← i + 1
    Fin Tant que
Fin Procédure

Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        Suivants ← suivants(nœud)
        j ← 0
        Tant que j < taille(Suivants)
        Début
            arc ← Suivants[j]
            VisiteArc(arc)
            j ← j + 1
        Fin Tant que
    Fin Si
Fin Procédure

Procédure VisiteArc (arc)
Début
    destination ← destination(arc)
    VisitNoeud(destination)
Fin Procédure
```

Exemple 2.5

Cet exemple permet de visualiser le parcours du graphe. A chacun des nœuds et des arcs est associé un numéro correspondant à l'ordre dans lequel ils sont parcourus.

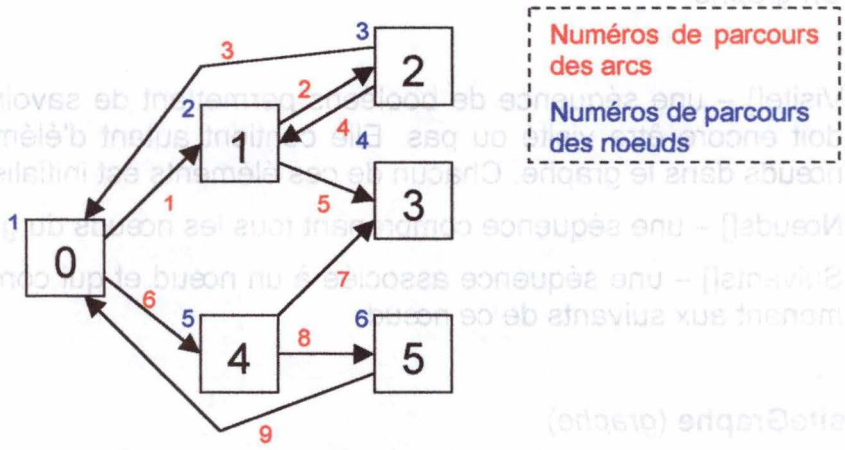


Figure 2.12 : Parcours d'un graphe quelconque.

2.4.6 *Calculs préalables au dessin du graphe*

A. Introduction

Avant de pouvoir dessiner le graphe, plusieurs calculs doivent être réalisés. En effet, pour rendre le dessin le plus lisible possible, il s'avère que plusieurs éléments doivent être pris en compte.

Afin de faciliter la compréhension du lecteur, ces calculs ont été regroupés en modules, selon leur utilité. Chaque module décrit un ou plusieurs algorithmes. Les modules de calcul sont au nombre de sept.

Tous les algorithmes de calcul ont la même structure, à savoir qu'ils sont basés sur l'algorithme de parcours d'un graphe décrit ci-dessus. Nous limiterons donc les explications concernant ces algorithmes aux explications concernant les modifications apportées à l'algorithme de parcours du graphe.

Le premier module sert à associer un niveau de dessin à chaque nœud du graphe. Le niveau de dessin correspond au niveau d'imbrication qu'occupe le traitement du nœud dans l'algorithme de parcours du graphe. L'utilité de ce niveau de dessin sera explicitée en temps voulu.

Dans le second module, nous définirons les différentes catégories d'arcs qui, selon nous, composent le graphe. Chacune de ces catégories regroupe un sous-ensemble d'arcs. Le dessin des arcs sera différent selon qu'ils appartiennent à l'une ou l'autre catégorie. À nouveau, l'algorithme de parcours d'un graphe joue un rôle primordial dans la détermination de la catégorie à laquelle appartient un arc.

Le troisième module a pour but de permettre le dessin des arcs parallèlement par rapport aux bords des boîtes. En effet, il s'agit de déterminer les couloirs qui doivent être ajoutés entre les nœuds afin d'y faire passer les segments horizontaux ou verticaux composant les arcs. De plus, nous déterminerons l'espace maximal à ajouter entre deux niveaux de dessin. En effet, pour améliorer la lisibilité du graphe, nous désirons aligner verticalement les nœuds d'un même niveau de dessin.

L'étape suivante consiste à calculer les dimensions du rectangle qui représente un nœud du graphe, c'est-à-dire de la boîte.

Il s'agit ensuite de calculer la *Bounding Box* associée à chaque nœud. Une *Bounding Box* est le plus petit rectangle qui contient un nœud et ses éventuels descendants. L'algorithme de parcours de graphe est, une fois de plus, au centre du problème. L'utilité de la *Bounding Box* sera détaillée au fur et à mesure des explications qui s'y rapportent. Cependant nous pouvons déjà affirmer qu'il s'agit d'un simple outil de dessin.

Enfin, les deux dernières étapes avant le dessin à proprement dit ont pour objectifs, pour la première, de déterminer le point d'encrage de chaque nœud, et, pour la seconde, d'assigner ses points de routage à chacun des arcs du graphe. Par point d'encrage, nous entendons le point du plan de dessin qui devra correspondre au coin supérieur gauche d'une boîte. Les points de routage sont les points par lesquels nous allons faire passer l'arc de manière à contourner les boîtes représentant les nœuds.

Exemple 2.6

Cet exemple permet de visualiser les *Bounding Boxes*, le point d'encrage d'un nœud et les points de routage d'un arc. Pour ce faire, nous avons repris l'exemple 2.3 en mettant ces éléments en évidence.

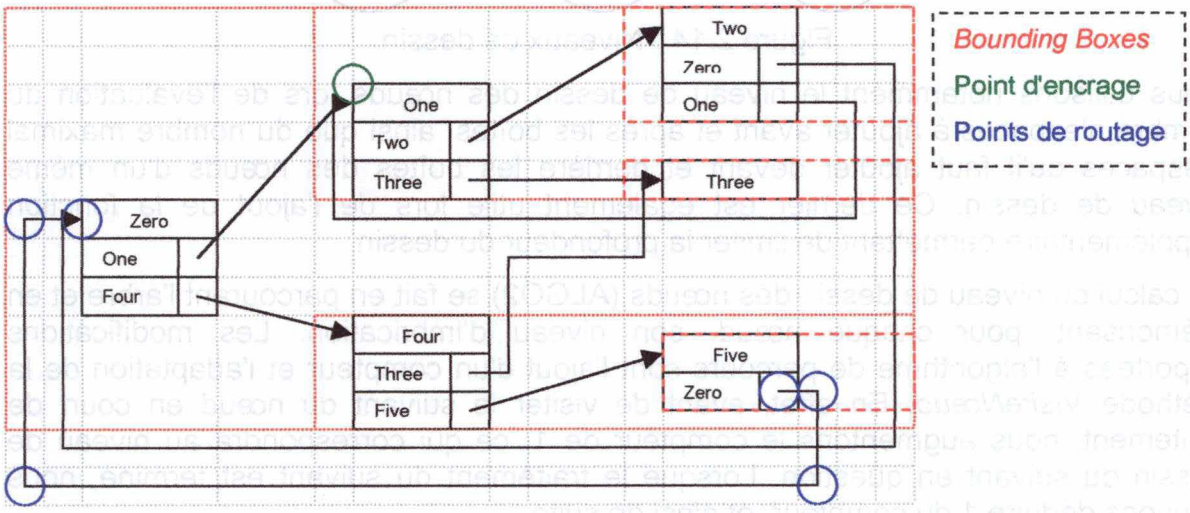


Figure 2.13 : Graphe quelconque avec la grille d'alignement.

Ces deux étapes ne peuvent être réalisées qu'en dernier lieu. En effet, les cinq premiers modules ont pour but de garnir les attributs des classes consacrées au dessin du graphe. C'est à partir de ces résultats que les coordonnées des points d'encrage et des points de routage sont calculées.

Lorsque ces deux étapes sont terminées, le graphe peut être dessiné.

B. Calcul du niveau de dessin de chaque nœud

Comme nous l'avons vu dans l'introduction, le niveau de dessin correspond au niveau d'imbrication qu'occupe le traitement d'un nœud dans le parcours du graphe.

Exemple 2.7

Cet exemple permet de visualiser ce que nous entendons par niveau de dessin.

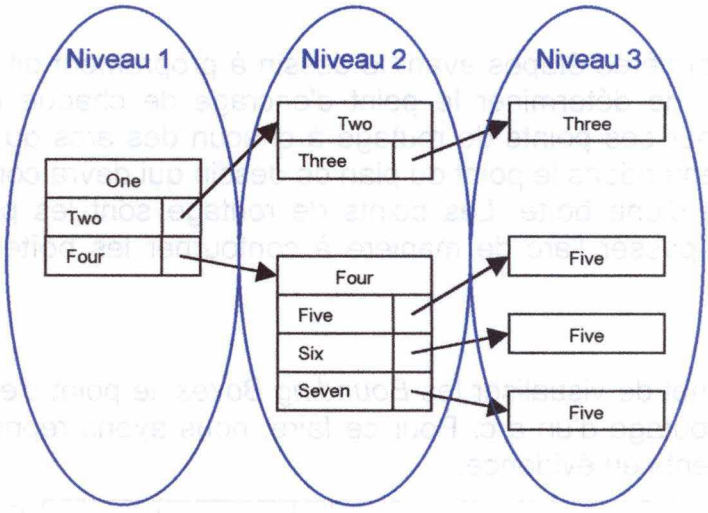


Figure 2.14 : Niveaux de dessin.

Nous utilisons notamment le niveau de dessin des nœuds lors de l'évaluation du nombre d'espaces à ajouter avant et après les boîtes, ainsi que du nombre maximal d'espaces qu'il faut ajouter devant et derrière les boîtes des nœuds d'un même niveau de dessin. Ce dernier est également utile lors de l'ajout de la fonction supplémentaire permettant de limiter la profondeur du dessin.

Le calcul du niveau de dessin des nœuds (ALGO2) se fait en parcourant l'arbre et en mémorisant, pour chaque nœud, son niveau d'imbrication. Les modifications apportées à l'algorithme de parcours sont l'ajout d'un compteur et l'adaptation de la méthode *VisiteNœud*. En effet, avant de visiter le suivant du nœud en court de traitement, nous augmentons le compteur de 1, ce qui correspondra au niveau de dessin du suivant en question. Lorsque le traitement du suivant est terminé, nous pouvons déduire 1 du compteur, et ainsi de suite.

Cet algorithme prévoit également de déterminer le nombre total des niveaux de dessin. Pour ce faire, nous ajoutons à nouveau un compteur permettant de déterminer, lors du traitement d'un nœud, la valeur maximale entre le niveau de dessin du nœud et le niveau de dessin maximal déjà rencontré.

ALGO2 – Calcul du niveau de dessin

Données : Un graphe.

Résultat : /

Utilise : nbrNiveaux – Un entier dont la valeur en fin de parcours du graphe est celle du nombre maximal de niveaux de dessin. Ce champ est initialisé à 0.

compteur – Un entier dont la valeur, à l'entrée de la méthode *VisiteNœud* de l'algorithme de parcours du graphe, est celle du niveau de dessin d'un nœud. Ce champ est initialisé à 0.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        niveau(noeud) ← compteur
        nbrNiveaux ← maximum(nbrNiveaux, compteur)
        Suivants ← suivants(nœud)
        j ← 0
        Tant que j < taille(Suivants)
        Début
            compteur ← compteur + 1
            arc ← Suivants[j]
            VisiteArc(arc)
            compteur ← compteur - 1
            j ← j + 1
        Fin Tant que
    Fin Si
Fin Procédure
```

C. Différencier les arcs

Afin de dessiner tous les arcs en respectant les conventions choisies, nous avons discerné deux catégories d'arcs : les arcs d'arbre et les arcs auxiliaires. Ces deux catégories sont définies ci-après. L'algorithme permettant de les distinguer suit ces définitions. L'article *[KDL]* nous a considérablement aidés dans la réalisation de cet algorithme, principalement la section concernant les graphes dirigés.

- ARCS D'ARBRE

Nous avons posé comme convention que les nœuds du graphe doivent être disposés selon une structure d'arbre définie par un parcours en profondeur d'abord. Or, comme nous l'avons vu pour le parcours du graphe, grâce à l'utilisation de la séquence de booléens permettant de déterminer si un nœud a déjà ou non été visité, les nœuds ne sont traités qu'une seule fois. Cependant, l'algorithme visite tous les arcs ; c'est seulement lorsqu'il visite un nœud qu'il vérifie si celui-ci a déjà été traité ou non. Afin de déterminer les sous-arbres à extraire du graphe (il y a un sous-arbre par composante simplement connexe), il semble nécessaire de déterminer les arcs qui le constituent, c'est-à-dire les arcs menant pour la première fois à un nœud lors du parcours du graphe.

En théorie des graphes, si nous considérons un graphe quelconque $G = (X, U)$, les arcs nécessaires au parcours du graphe forment un sous-ensemble d'arcs $V \subset U$. Nous pouvons donc affirmer que l'arbre défini par l'algorithme de parcours du graphe est un graphe partiel H du graphe à dessiner G. Les arcs le constituant sont appelés *arcs d'arbre*. L'ensemble de ces arcs forme la première catégorie d'arcs constituant le graphe.

Exemple 2.8

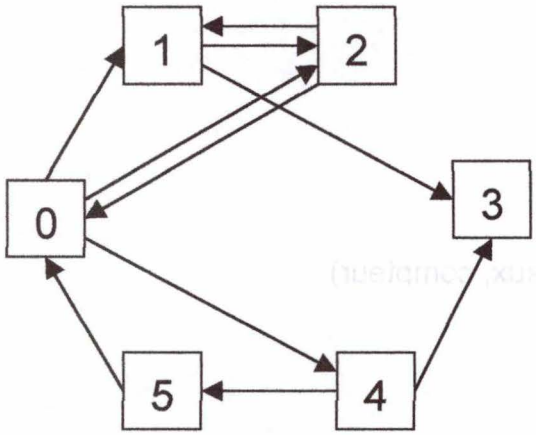


Figure 2.15 : G – Graphe entier.

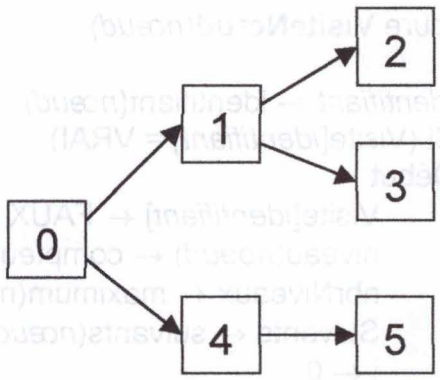


Figure 2.16 : H – Graphe partiel ne comprenant que les arcs d'arbre.

- ARCS AUXILIAIRES

Les arcs auxiliaires sont les arcs qui ne sont pas des arcs d'arbre.

Nous avons distingué deux sous-catégories d'arcs auxiliaires : les *arcs arrières* et les *arcs transversaux*. Un arc arrière est un arc qui a pour destination un nœud appartenant à l'ensemble des ascendants de son nœud origine. Les arcs transversaux sont tous les autres arcs, c'est-à-dire ceux qui n'entrent ni dans la catégorie des arcs principaux ni dans la sous-catégorie des arcs arrières. On les appelle arcs transversaux en référence au fait qu'ils peuvent traverser le dessin de part en part pour relier deux nœuds

Exemple 2.9

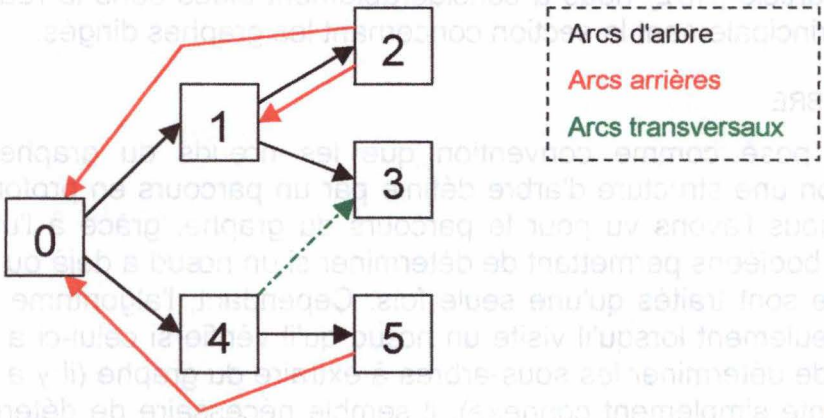


Figure 2.17 : Arcs auxiliaires.

- TYPE D'ARCS ET ALGORITHME

Pour des raisons pratiques, nous avons attribué un nom anglophone à chacune des catégories et des sous-catégories définies ci-dessus. Ces noms correspondent aux types associés à chacun des arcs par l'algorithme de différenciation des arcs (ALGO3). Les arcs d'arbre sont rassemblés sous le nom d'arcs FORWARD ; les arcs arrières sont appelés arcs BACK ; et les arcs transversaux portent le nom de CROSS.

Les modifications apportées à l'algorithme de parcours sont l'ajout d'une séquence permettant d'associer une "couleur" à chaque nœud. Cette couleur sera BLANC si le nœud n'a pas encore été visité, GRIS s'il a été visité mais que le traitement de ses descendants est en cours, et NOIR si tous les descendants du nœud ont été traités. Cette séquence remplace la séquence Visite utilisée habituellement lors du parcours du graphe. La méthode *VisiteNœud* est donc sensiblement modifiée pour supporter ce changement.

De plus, il convient d'adapter la méthode *VisiteArc* afin d'assigner à chacun des arcs le type qui lui correspond. Le type d'un arc dépend en fait de la couleur de sa destination. En effet, si la couleur associée à la destination de l'arc dont le traitement est en cours est NOIR, le type de l'arc sera CROSS car il s'agit d'un nœud dont le traitement a déjà eu lieu et qui n'est pas un ascendant de l'origine de l'arc. Si la couleur est GRIS, l'arc sera du type BACK puisqu'il s'agit d'un nœud dont le traitement n'est pas terminé, c'est-à-dire que le nœud destination de l'arc est un ascendant du nœud origine. Il s'agit bien, dans ce cas, d'un arc arrière. Enfin, si la couleur du nœud est BLANC, le nœud n'a pas encore été visité, et l'arc dont le traitement est en cours est un arc d'arbre.

ALGO3 – Différenciation des arcs

Données : Un graphe.

Résultat : /

Utilise : Couleur[] – une séquence d'entiers (BLANC = 0, GRIS = 2, NOIR = 3), permettant de savoir si un nœud a déjà été visité ou pas, et si ses descendants ont été traités ou non. Elle contient autant d'éléments que de nœuds dans le graphe. Chacun de ces éléments est initialisé à BLANC.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Couleur[identifiant] = BLANC)
    Début
        Couleur [identifiant] ← GRIS
        Suivants ← suivants(nœud)
        i ← 0
        Tant que i < taille(Suivants)
        Début
            arc ← Suivants[i]
            VisiteArc(arc)
            i ← i + 1
        Fin Tant que
        Couleur [identifiant] = NOIR
    Fin Si
Fin Procédure
```

```
Procédure VisiteArc (arc)
Début
    destination ← destination(arc)
    identifiant ← identifiant(destination)
    Si (Couleur[identifiant] = NOIR)
        type(arc) ← CROSS
    Si (Couleur[identifiant] == GRIS)
        type(arc) ← BACK
    Si non
        type(arc) ← FORWARD
    Fin Si
    VisitNœud(destination)
Fin Procédure
```

Exemple 2.10

Cet exemple permet de visualiser les différentes catégories d'arcs sur un dessin semblable à ce qui est produit par l'utilitaire d'affichage.

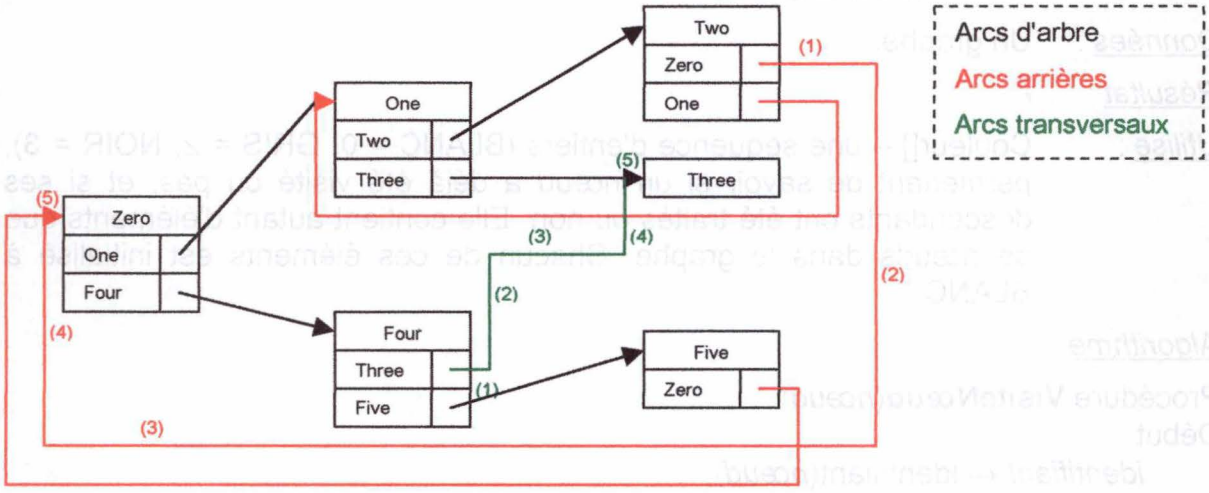


Figure 2.18 : Mise en évidence des arcs principaux.

Dans la suite, lorsque nous parlerons du premier segment d'un arc, il s'agira du segment horizontal dessiné à la sortie du nœud origine (1), le deuxième segment d'arc est le segment vertical attaché au premier (2), le troisième est le segment horizontal suivant (3), le quatrième correspond à l'autre segment vertical (4) et le cinquième est le dernier segment dont l'extrémité est la flèche correspondant à l'extrémité finale de l'arc (5).

D. Déterminer les couloirs pour le routage des arcs

Pour respecter les conventions selon lesquelles nous avons choisi qu'aucun arc ne peut traverser un nœud et que les arcs ne peuvent pas se recouvrir, nous allons devoir mettre en place tout ce qui est nécessaire au routage de certains arcs. En effet, il est indispensable de prévoir le fait que les arcs auxiliaires doivent contourner les boîtes et cela sans se superposer. Ces arcs seront donc dessinés de manière parallèle aux côtés des boîtes représentant les nœuds.

Il ne s'agit pas encore de calculer les points de routage mais bien de déterminer les couloirs qui doivent être ajoutés entre les boîtes afin de permettre le passage des segments horizontaux ou verticaux composant les arcs. De plus, nous déterminerons l'espace maximal à ajouter entre deux niveaux de dessin de manière à aligner verticalement les nœuds d'un même niveau de dessin.

Les couloirs sont en fait la distance qu'il est nécessaire de prévoir entre les boîtes afin de permettre le passage des arcs. Il s'agit donc de calculer ces distances et, lors d'une étape ultérieure, d'en tenir compte pour le placement des arcs et des boîtes. Pour faciliter les calculs, nous avons assigné une largeur constante à l'espace qu'il est nécessaire d'ajouter pour dessiner les segments d'arc, c'est-à-dire aux couloirs.

Ce module se compose de cinq algorithmes permettant de déterminer les éléments suivants : le nombre de couloirs à ajouter après la boîte correspondant à l'origine de l'arc, le nombre de couloir à ajouter avant la boîte correspondant à la destination de l'arc, celui à ajouter sous la *Bounding Box* du nœud sous lequel le troisième segment de l'arc doit passer, et l'espace maximal à ajouter avant et après les boîtes correspondant aux nœuds d'un même niveau de dessin.

- COULOIRS À AJOUTER APRÈS UNE BOÎTE

Pour déterminer le nombre de couloirs à ajouter après une boîte, il faut tenir compte de plusieurs éléments.

Tout d'abord, afin de limiter les croisements d'arcs à la sortie d'une boîte, nous avons décidé que le deuxième segment de l'arc auxiliaire, dont l'origine est située au niveau du premier attribut, serait le plus éloigné de la boîte, que celui correspondant au deuxième attribut se rapprocherait un peu et ainsi de suite jusqu'au dernier dont le deuxième segment serait le moins éloigné de la boîte.

Ce raisonnement est également valable sur l'ensemble des attributs des suivants d'un nœud, c'est-à-dire que le deuxième segment de l'arc auxiliaire, ayant pour origine le premier attribut de la boîte correspondant au premier suivant du nœud en cours de traitement, est le plus éloigné de cette boîte, alors que celui du dernier attribut de la boîte correspondant au dernier suivant est le plus proche de sa boîte. C'est pour cette raison que l'algorithme calculant le nombre de couloirs à ajouter après une boîte considère les suivants du nœud en commençant par le dernier. En effet, de cette manière, il tient compte des couloirs ajoutés après les suivants dessinés en dessous du suivant en cours de traitement.

Ainsi, au fur et à mesure que l'algorithme visite chaque nœud du graphe, il parcourt, en commençant par le dernier, la séquence des suivants du nœud. Pour chaque suivant, il calcule le nombre de couloirs à ajouter après la boîte qui le représente et y ajoute la somme des couloirs à ajouter après les suivants déjà traités.

Exemple 2.11

Dans l'exemple qui suit, lors du traitement du nœud *One*, l'algorithme détermine le nombre de couloirs à ajouter après ses suivants, c'est-à-dire *Two* et *Three*. Il commence par compter les couloirs à ajouter après le nœud *Three*, dont le nombre est 1 puisqu'il y a un arc arrière. Ensuite, il s'occupe du nœud *Two* après lequel il doit ajouter 2 couloirs pour les arcs arrières dont il est l'origine. Enfin, il ajoute à ce nombre la somme des couloirs à ajouter après les suivants déjà traités, en l'occurrence 1, puisque le nœud déjà traité est *Three*.

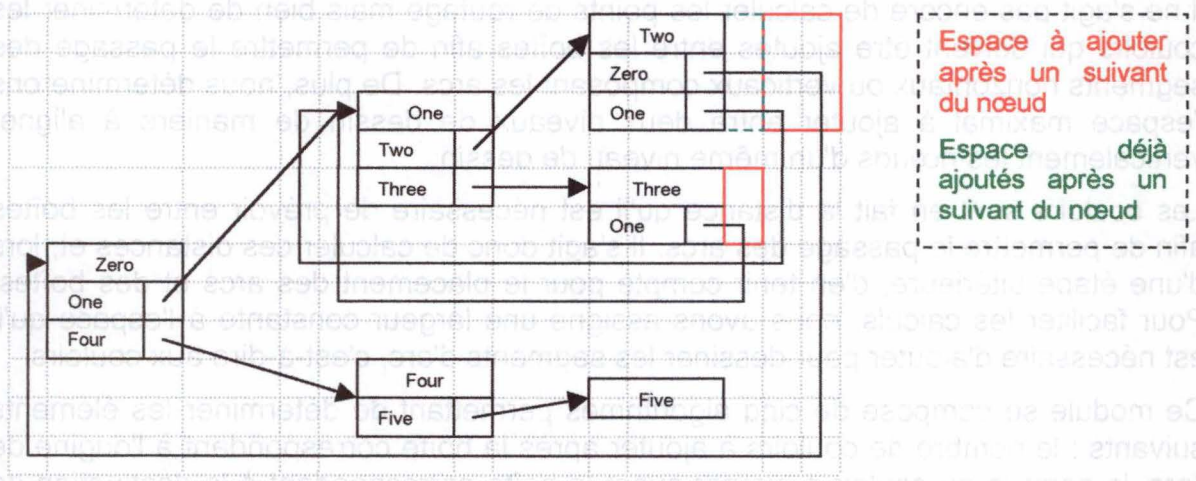


Figure 2.19 : Couloirs à ajouter après les suivants d'un nœud.

De plus, les nœuds d'un même niveau de dessin peuvent être chacun l'origine d'arcs auxiliaires, d'où l'importance de mémoriser, pour chaque niveau de dessin, la somme des nombres de couloirs à ajouter après les nœuds déjà traités du niveau. A chaque traitement d'un suivant du nœud visité, l'algorithme additionne la somme correspondant au niveau de dessin du suivant au nombre de couloirs déterminés pour ce suivant.

Exemple 2.12

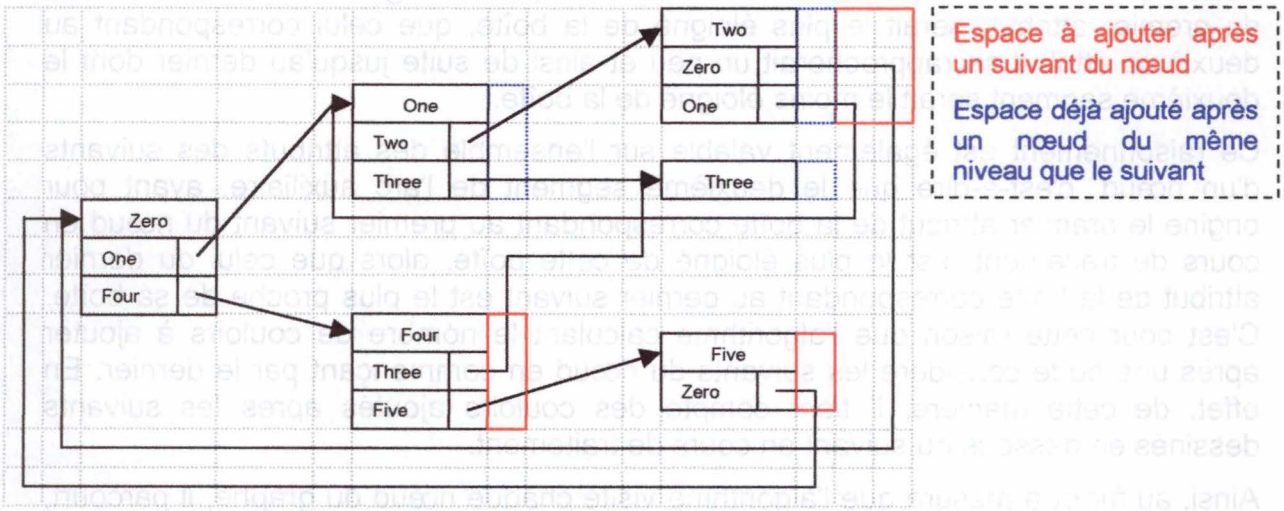


Figure 2.20 : Couloirs à ajouter après les boîtes.

Les modifications à apporter à l'algorithme de parcours du graphe sont tout d'abord deux attributs supplémentaires : une séquence de booléens permettant de déterminer si un nœud est la racine d'un sous-arbre déterminé par l'algorithme de parcours ou pas, et une séquence d'entiers contenant pour chaque niveau de dessin le nombre de couloirs déjà ajoutés après les nœuds déjà traités du niveau.

La procédure *VisiteGraphe* est modifiée en deux points. Tout d'abord, c'est lors de la visite du graphe que la première séquence est garnie. Ensuite, une procédure est appelée afin de calculer le nombre de couloirs à ajouter après les racines. En effet, lors du traitement d'un nœud, l'algorithme calcule le nombre de couloirs à ajouter après chacun de ses suivants, les racines ne sont donc pas prises en compte à ce niveau.

La procédure *VisiteNœud* n'est modifiée que par l'appel d'une procédure calculant le nombre de couloirs à ajouter après le nœud visité.

Plusieurs procédures sont ainsi ajoutées à l'algorithme, l'une calculant le nombre de couloirs à ajouter après la racine d'un sous-arbre, l'autre calculant celui à ajouter après les autres nœuds, et une dernière déterminant le nombre d'arcs auxiliaires issus d'un nœud.

ALGO4 – Déterminer le nombre de couloirs à ajouter après une boîte

Données : Un graphe, le nombre de niveaux de dessin du graphe.

Résultat : /

Utilise : CouloirNiveau[] – une séquence d'entiers contenant pour chaque niveau de dessin le nombre de couloirs déjà ajoutés après les nœuds du niveau. Il y a autant d'éléments qu'il y a de niveaux de dessin dans le graphe, le nombre de niveau de dessin étant reçu en argument. Chaque élément de la séquence est initialisé à 0.

Racine[] - une séquence de booléens permettant de déterminer si un nœud est la racine d'un sous-arbre ou pas. Il y a autant d'éléments que de nœuds dans le graphe. Ils sont tous initialisés à FAUX.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    Nœuds ← nœuds(graphe)
    i ← 0
    Tant que i < taille(Nœuds)
    Début
        nœud ← Nœuds[i]
        identifiant ← identifiant(nœud)
        Si (Visite[identifiant] = VRAI)
            Racine[identifiant] ← VRAI
        VisiteNœud(nœud)
        CalculerCouloirsAprèsRacine(nœud)
        i ← i + 1
    Fin Tant que
Fin Procédure

Procédure CalculerCouloirsAprèsRacine(nœud)
Début
    Si (Racine[nœud] = VRAI)
    Début
        nbrArcsSortant ← CalculerArcsSortant(nœud)
        couloirsAprès(nœud) ← nbrArcsSortant
    Fin Si
Fin Procédure
```

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        ...
        CalculerCouloirsAprès(nœud)
    Fin Si
Fin Procédure

Procédure CalculerCouloirsAprès(nœud)
Début
    Suivants ← suivants(nœud)
    i ← taille(Suivants)
    Tant que i > 0
    Début
        arc ← Suivants[i]
        Si (type(arc) = FORWARD)
        Début
            destination ← destination(arc)
            nbrArcsSortant ← CalculerArcsSortant(destination)
            couloirsAprès(nœud) ← CouloirsNiveau[niveau(nœud)]
                                  + nbrArcsSortant
            CouloirsNiveau[niveau(nœud)] ←
                CouloirsNiveau[niveau(nœud)] + nbrArcsSortant
        Fin Si
        i ← i - 1
    Fin Tant que
Fin Procédure

Fonction CalculerArcsSortant(nœud)
Début
    arcsSortant ← 0
    Suivants ← suivants(nœud)
    i ← 0
    Tant que i < taille(Suivants)
    Début
        arc ← Suivants[i]
        Si (type(arc) <> FORWARD)
            arcsSortant ← arcsSortant + 1
        i ← i + 1
    Fin Tant que
    Retourner arcsSortant
Fin Procédure
```

- COULOIRS À AJOUTER AVANT UNE BOÎTE

Pour calculer le nombre de couloirs à ajouter avant une boîte, la démarche est semblable à celle utilisée pour les couloirs à ajouter après les boîtes, mis à part qu'il n'est pas utile dans ce cas de commencer par le dernier suivant du nœud en cours de traitement. En effet, les points de routage seront assignés aux couloirs dans l'ordre de leur dessin. Le premier couloir en commençant par la gauche se verra assigner le premier arc auxiliaire, le deuxième couloir prévu contiendra le deuxième arc auxiliaire, et ainsi de suite. Dans le cas des couloirs à ajouter après les boîtes, c'est le dernier couloir qui hébergeait le premier arc.

Exemple 2.13

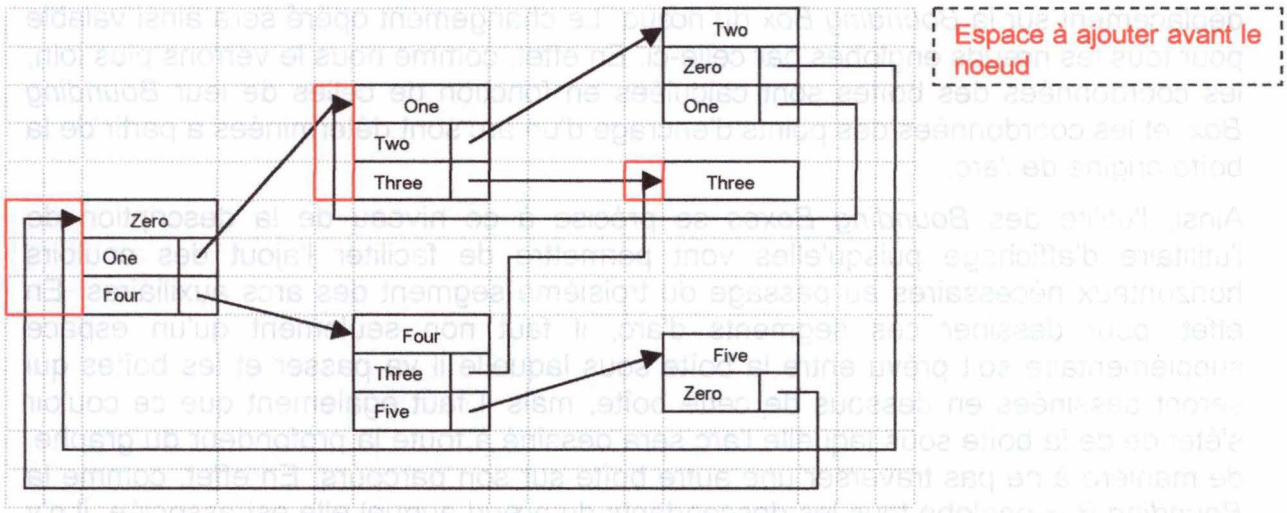


Figure 2.21 : Couloirs à ajouter avant les boîtes.

L'algorithme est semblable à celui permettant de déterminer le nombre de couloirs à ajouter après une boîte.

Cependant, la procédure calculant le nombre d'arcs auxiliaires dont le nœud est l'origine est remplacée par une procédure calculant le nombre d'arcs auxiliaires dont le nœud est la destination.

De plus, la séquence permettant de savoir, pour chaque niveau de dessin, le nombre de couloirs déjà ajoutés après les nœuds du niveau devient une séquence permettant de savoir, pour chaque niveau de dessin, le nombre de couloirs déjà ajoutés avant les nœuds du niveau.

Enfin, la séquence des suivants du nœud visité n'est plus parcourue en commençant par le dernier suivant mais bien par le premier.

Étant donné les ressemblances avec l'algorithme précédent, il ne semble pas nécessaire de spécifier précisément l'algorithme. Le lecteur peut se référencer à l'algorithme ALGO4 pour avoir une idée plus précise de ce dont il est question.

- COULOIRS À AJOUTER SOUS UNE *BOUNDING BOX*

Avant de déterminer le nombre de couloirs qu'il est nécessaire d'ajouter sous chacune des *Bounding Boxes*, nous allons rappeler ce que nous entendons par Bounding Box.

Une *Bounding Box* est le plus petit rectangle, dont les côtés sont parallèles aux axes X et Y du système de coordonnées, qui contient un nœud et ses éventuels descendants sur le sous-arbre déterminé par l'algorithme de parcours de graphe.

Il s'agit donc de regrouper un ensemble de nœuds du graphe dans un rectangle afin de ne pas avoir à traiter chacun de ces nœuds séparément. En effet, lorsqu'il s'agit de faire supporter à un nœud et à tous ses descendants le même déplacement, comme une translation horizontale ou verticale, il est plus facile de reporter ce déplacement sur la *Bounding Box* du nœud. Le changement opéré sera ainsi valable pour tous les nœuds englobés par celle-ci. En effet, comme nous le verrons plus loin, les coordonnées des boîtes sont calculées en fonction de celles de leur *Bounding Box*, et les coordonnées des points d'encrage d'un arc sont déterminées à partir de la boîte origine de l'arc.

Ainsi, l'utilité des *Bounding Boxes* se précise à ce niveau de la description de l'utilitaire d'affichage puisqu'elles vont permettre de faciliter l'ajout des couloirs horizontaux nécessaires au passage du troisième segment des arcs auxiliaires. En effet, pour dessiner ces segments d'arc, il faut non seulement qu'un espace supplémentaire soit prévu entre la boîte sous laquelle il va passer et les boîtes qui seront dessinées en dessous de cette boîte, mais il faut également que ce couloir s'étende de la boîte sous laquelle l'arc sera dessiné à toute la profondeur du graphe, de manière à ne pas traverser une autre boîte sur son parcours. En effet, comme la *Bounding Box* englobe tous les descendants du nœud auquel elle est associée, il n'y a pas de risque qu'un de ces nœuds ne soit traversé par un arc. Lors du calcul des coordonnées des boîtes, l'algorithme tiendra compte du fait que des couloirs ont été ajoutés sous certaines *Bounding Boxes*.

Dans le cas des arcs à ajouter sous les *Bounding Boxes*, il est nécessaire de savoir, pour un arc, le nœud sous la boîte duquel passe son troisième segment, et pour un nœud, les arcs qui passent sous sa boîte. Ces informations peuvent être réunies dans une matrice M de booléens dont les indices de lignes correspondent aux arcs et les indices de colonnes correspondent aux identifiants des nœuds. En effet, si l'arc i passe sous le nœud j, l'élément M[i, j] de la matrice est à VRAI, si non, il est à FAUX.

Comme il y a deux sous-catégories d'arcs auxiliaires, il y a deux cas à envisager. Le premier est celui des arcs arrières et le second est celui des arcs transversaux. Ces deux cas sont détaillés ci-après.

- *Arcs arrières*

Lorsqu'il s'agit d'un arc arrière, le couloir est ajouté sous la *Bounding Box* du nœud destination de l'arc. L'algorithme ajoute ainsi sous une *Bounding Box* autant de couloirs qu'il y a d'arcs arrières passant sous elle.

Exemple 2.14

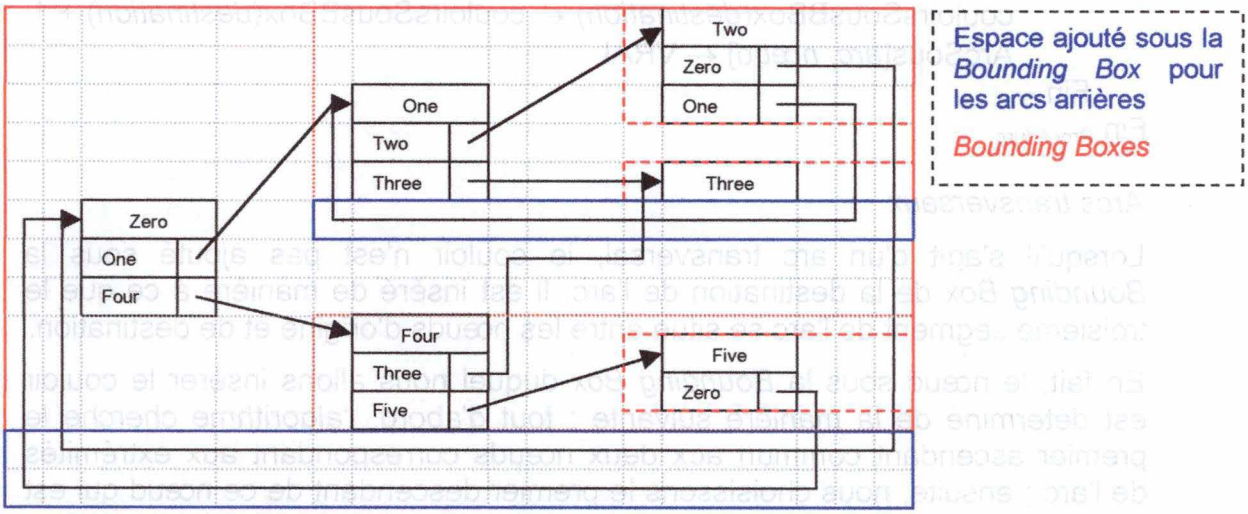


Figure 2.22 : Couloirs à ajouter sous les *Bounding Boxes* pour les arcs arrières.

L'algorithme de parcours d'un graphe ne subit dans ce cas qu'une modification. La procédure *VisiteArc* fait appel à une procédure supplémentaire. Cette procédure permet de déterminer le nombre de couloirs qui devront être ajoutés sous la destination de l'arc visité.

ALGO5 – Déterminer le nombre de couloirs à ajouter sous une *Bounding Box* pour les arcs arrières

Données : Un graphe.

Résultat : ArcSous – la matrice associant chaque arc au nœud sous lequel il passe.

Utilise : ArcSous[arc, nœud] – matrice de booléens permettant de mémoriser pour un nœud les arcs qui passent sous sa boîte. Il s'agit donc de mettre l'élément correspondant à l'arc et au nœud à VRAI dans le cas où cet arc passe sous ce nœud. Cette matrice a autant de lignes qu'il y a d'arcs et autant de colonnes qu'il y a de nœuds dans le graphe. Tous les éléments sont initialisés à FAUX.

Algorithme :

```
Procédure VisiteArc (arc)
Début
    destination ← destination(arc)
    CalculerCouloirsSousDestination(arc)
    VisitNœud(destination)
Fin Procédure
```

```
Procédure CalculerCouloirsSousDestination(arc)
Début
    destination ← destination(arc)
    origine ← origine(arc)
    Si (type(arc) = BACK)
    Début
        couloirsSousBBox(destination) ← couloirsSousBBox(destination) + 1
        ArcSous[arc, nœud] ← VRAI
    Fin Si
Fin Procédure
```

- *Arcs transversaux*

Lorsqu'il s'agit d'un arc transversal, le couloir n'est pas ajouté sous la *Bounding Box* de la destination de l'arc. Il est inséré de manière à ce que le troisième segment de l'arc se situe entre les nœuds d'origine et de destination.

En fait, le nœud sous la *Bounding Box* duquel nous allons insérer le couloir est *déterminé de la manière suivante : tout d'abord, l'algorithme cherche le* premier ascendant commun aux deux nœuds correspondant aux extrémités de l'arc ; ensuite, nous choisissons le premier descendant de ce nœud qui est sur le chemin menant à l'extrémité de l'arc (origine ou destination) dont l'identifiant du nœud est le plus petit. C'est sous la *Bounding Box* de ce nœud que le couloir sera ajouté.

Exemple 2.15

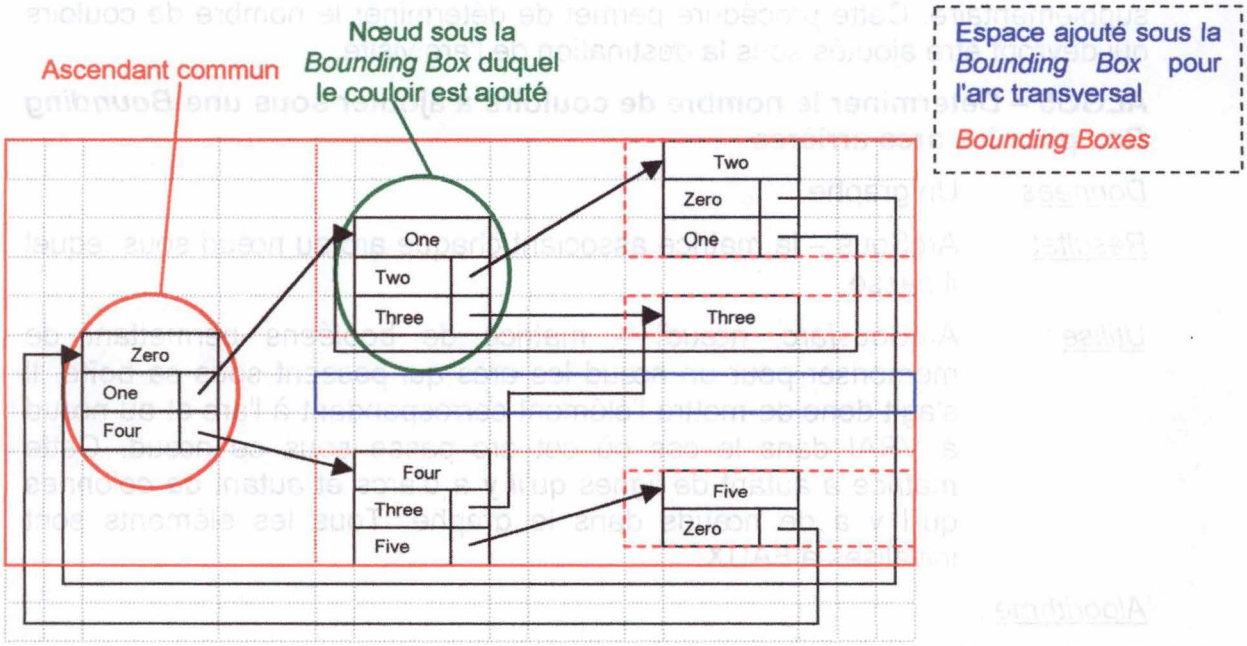


Figure 2.23 : Couloirs à ajouter sous les *Bounding Boxes* pour les arcs transversaux.

Dans ce cas, plusieurs éléments s'ajoutent à ceux utiles au parcours du graphe et à la matrice déjà utilisée dans l'algorithme précédent. Ces éléments sont nécessaires à l'identification de l'ascendant commun des deux extrémités d'un arc.

Tout d'abord, nous avons besoin d'un booléen permettant de savoir si l'ascendant commun a été trouvé ou non.

Ensuite, une séquence est utilisée dans le but de mémoriser les ascendants d'un nœud particulier. En effet, dans un premier temps, l'algorithme détermine quels sont les ascendants de l'origine de l'arc transversal dont il est question. Pour cela, il remonte l'arbre de parcours à partir de cette origine jusqu'à la racine de l'arbre, et assigne la valeur VRAI à chacun des éléments de la séquence correspondant à un nœud rencontré sur son parcours. Ensuite, en partant de la destination de l'arc, il remonte à nouveau le graphe et visite ainsi les ascendants de ce nœud. Pour chacun de ceux-ci, il vérifie la valeur de la séquence précédemment garnie afin de voir si l'ascendant de la destination n'est pas également un ascendant de l'origine, ce qui correspondrait au premier ascendant commun recherché.

Enfin, la séquence permettant de mémoriser, pour chaque nœud, s'il est une racine du sous-arbre correspondant au parcours du graphe est à nouveau utilisée.

La procédure *VisiteGraphe* n'est modifiée qu'en un point, elle permet de déterminer si un nœud est une racine de l'arbre ou non. Il s'agit ainsi de la même procédure que celle utilisée dans l'algorithme ALGO4, sans l'appel à la méthode déterminant le nombre de couloirs à ajouter après les racines des sous-arbres du graphe.

La procédure VisiteArc est complétée de tous les éléments qui permettent de déterminer où et comment ajouter les couloirs sous la Bounding Box. Pour ce faire, elle fait appel à deux procédures, l'une permettant de retrouver l'ascendant commun de l'origine et de la destination de l'arc visité, et l'autre choisissant le descendant du nœud commun sous la *Bounding Box* duquel elle ajoute le couloir.

ALGO6 – Déterminer le nombre de couloirs à ajouter sous une *Bounding Box* pour les arcs transversaux

Données : Un graphe.

Résultat : ArcSous – la matrice associant chaque arc au nœud sous lequel il passe.

Utilise : trouvé – booléen permettant de savoir si l'ascendant commun de l'origine et de la destination de l'arc visité à été découvert. Cet élément est initialisé à FAUX.

Ascendants[] – séquence de booléens permettant de savoir quels sont les ascendants de l'origine de l'arc visité. Les éléments de cette séquence sont initialisés à FAUX.

commun – ascendant commun aux deux extrémités de l'arc visité. Cet élément est initialisé à NULL.

fin – booléen utilisé dans la procédure de calcul des couloirs à ajouter sous la Bounding Box.

Algorithme :

```
Procédure VisiteArc (arc)
Début
    destination ← destination(arc)
    VisitNœud(destination)
    Si (type(arc) = CROSS)
    Début
        commun ← CalculerAscendantCommun(arc)
        Si (commun <> NULL)
            CalculerCouloirsSous(commun, arc)
        Remettre tous les éléments de Ascendants à FAUX
        trouvé ← FAUX
    Fin Si
Fin Procédure

Fonction commun CalculerAscendantCommun(arc)
Début
    origine ← origine (arc)
    Caluler(origine)
    destination ← destination(arc)
    commun ← Caluler(destination)
    Renvoyer commun
Fin Procédure

Procédure CalculerCouloirsSous(commun, arc)
Début
    fin ← FAUX
    Suivants ← suivants(commun)
    i ← 0
    Tant que (i < taille(Suivants)) & (fin = FAUX)
    Début
        suivant ← Suivants[i]
        destination ← destination(suivant)
        identifiant ← identifiant(destination)
        Si (type(suivant) = FORWARD) & (Ascendants[identifiant])
        Début
            couloirsSousBBox(destination)
                                ← couloirsSousBBox(destination) + 1
            arcSous[arc, destination] ← VRAI
            fin ← VRAI
        Fin Si
        i ← i + 1
    Fin Tant que
Fin Procédure
```

```
Fonction commun Calculer(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Racine[identifiant] = VRAI)
    Début
        Ascendants[identifiant] ← VRAI
        trouvé ← VRAI
        commun ← nœud
        Renvoyer commun
    Fin Si
    Si (Ascendants[identifiant] = VRAI)
    Début
        trouvé ← VRAI
        commun ← nœud
        Renvoyer commun
    Fin Si
    Sinon
        Ascendants[identifiant] ← VRAI
        Précédents ← précédents(nœud)
        i ← 0
        Tant que i < taille(Précédents)
        Début
            précédent ← Précédents[i]
            Si (type(arc) = FORWARD)
            Début
                origine ← origine(précédent)
                commun ← Calculer(origine)
                Si (trouvé = VRAI)
                    Renvoyer commun
            Fin Si
            i ← i + 1
        Fin Tant que
        Renvoyer NULL
    Fin Sinon
Fin Procédure
```

- MAXIMUM DE COULOIRS À AJOUTER AVANT ET APRÈS UNE BOÎTE

Dans le but d'aligner tous les nœuds d'un même niveau de dessin, et de s'assurer qu'aucun arc ne traverse un nœud, il s'avère nécessaire de calculer, pour chaque niveau de dessin, le nombre maximal de couloirs à ajouter avant et après tous les nœuds du niveau de dessin. Il suffit dans ce cas de parcourir le graphe et de déterminer, pour chaque niveau de dessin, le maximum parmi les nombres de couloirs à ajouter avant les nœuds du niveau en question, et de même pour ceux à ajouter après.

L'algorithme de parcours du graphe se voit ajouter deux séquences, l'une pour mémoriser le nombre de couloirs à ajouter après chaque niveau et l'autre pour ceux à ajouter avant.

De plus, avant de sortir du test de la procédure *VisiteNœud*, il calcule le nombre maximal de couloirs à ajouter avant et après chaque niveau de dessin et les assigne à l'élément correspondant dans chacune des séquences.

Lorsque les deux séquences sont complétées, un nouveau parcours est nécessaire, à la fin de la procédure *VisiteGraphe*, ayant pour but l'assignation de ces valeurs à chacun des nœuds.

ALGO7 – Déterminer le nombre maximal de couloirs à ajouter avant et après les boîtes de chaque niveau

Données : Un graphe, le nombre de niveaux de dessin du graphe.

Résultat : /

Utilise : MaxAvant – séquence d'entiers permettant de mémoriser le nombre de couloirs à ajouter devant la boîte de chaque nœud. Cette séquence se compose d'autant d'éléments qu'il y a de niveau de dessin dans le graphe, cet élément étant reçu en argument. Ils sont tous initialisés à 0.

MaxAprès – séquence semblable à la précédente, mais pour les couloirs à ajouter après.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    ...
    i ← 0
    Tant que i < taille(Nœuds)
    Début
        nœud ← Nœuds[i]
        niveau ← niveau(nœud)
        maxAvant(nœud) ← MaxAvant[niveau]
        maxAprès(nœud) ← MaxAprès[niveau]
        i ← i + 1
    Fin Tant que
Fin Procédure

Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        ...
        niveau ← niveau(nœud)
        MaxAvant[niveau] ← max(MaxAvant[niveau], CouloirsAvant(nœud))
        MaxAprès[niveau] ← max(MaxAprès[niveau], CouloirsAprès(nœud))
    Fin Si
Fin Procédure
```

E. Calculer les dimensions des boîtes

Comme nous l'avons déjà vu, une boîte se compose de plusieurs compartiments : le premier correspondant au nom de variable ou à la valeur d'un élément selon qu'il s'agit de l'environnement ou du store, et les autres compartiments correspondant aux éventuels attributs de l'objet représenté.

Nous savons déjà que la largeur des boîtes est fixée et identique pour chacune d'elles, il est donc inutile de la calculer. De plus, la hauteur d'un compartiment est également fixée et a donc une valeur constante.

Il reste cependant à calculer la hauteur des boîtes, puisque celle-ci dépend du nombre d'attributs de l'objet représenté par la boîte. Pour cela, il suffit de parcourir le graphe et, pour chaque nœud, de multiplier la hauteur d'un compartiment par le nombre d'attributs + 1. D'où, la seule procédure de l'algorithme de parcours de graphe qui est modifiée est *VisiteNœud*. En effet, c'est à cette procédure qu'est attribuée la tâche de déterminer, pour chaque nœud, la hauteur de la boîte qui le représente.

ALGO8 – Calculer la hauteur d'une boîte

Données : Un graphe.

Résultat : /

Utilise : hauteur – un entier permettant de mémoriser la hauteur de la boîte au fur et à mesure des calculs.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        hauteur ← 0
        Suivants ← suivants(nœud)
        Si taille(Suivants) = 0
            hauteur ← hauteurCompartiment
        Sinon
        Début
            j ← 0
            Tant que j < taille(Suivants)
            Début
                arc ← Suivants[j]
                VisiteArc(arc)
                hauteur ← (nbrAttributs(nœud) + 1) * hauteurCompartiment
                j ← j + 1
            Fin Tant que
        Fin Sinon
        hauteurBoîte(nœud) ← hauteur
    Fin Si
Fin Procédure
```

F. Calculer les dimensions des *Bounding Boxes*

Comme nous l'avons vu plus haut, la *Bounding Box* regroupe un nœud et les descendants de ce nœud sur le sous-arbre déterminé par l'algorithme de parcours du graphe. Elle a pour but de faciliter les calculs des coordonnées de boîte et, par la suite, des points de routage des arcs.

Dès cette étape, la *Bounding Box* associée à une boîte tient compte des couloirs à ajouter avant et après chaque boîte. Par contre, comme son appellation l'indique, un *couloir à ajouter sous une Bounding Box n'est pas inclus dans cette Bounding Box.*

De manière générale, un couloir vertical de taille fixe est toujours ajouté entre les boites, et un couloir horizontal est également prévu entre les *Bounding Boxes*. En effet, même si le graphe ne comporte aucun arc auxiliaire, il faut prévoir les espaces permettant de dissocier les boîtes.

Exemple 2.16

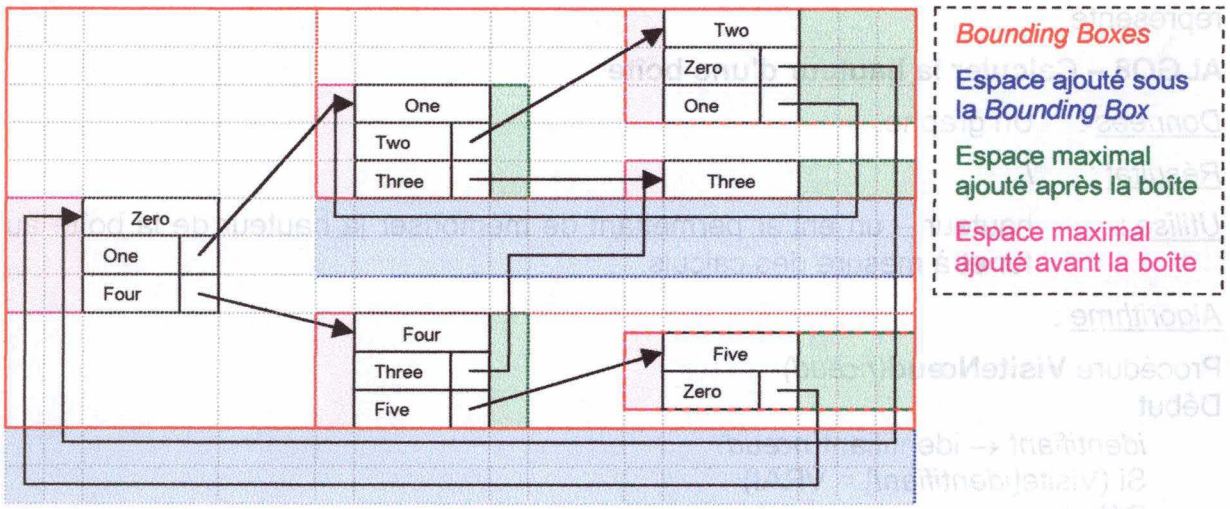


Figure 2.24 : Dimensions d'une Bounding Box.

Les dimensions des *Bounding Boxes* associées aux nœuds du graphe sont déterminées en deux fois. Un algorithme permet de calculer la hauteur de chacune des *Bounding Boxes* et un autre permet d'en calculer la largeur. Ces deux algorithmes sont une fois de plus calqués sur celui de parcours du graphe. C'est la procédure *VisiteNœud* de cet algorithme qui se voit assigner cette tâche.

- HAUTEUR DES *BOUNDING BOXES*

La hauteur d'une *Bounding Box* est déterminée en additionnant les hauteurs des *Bounding Boxes* qu'elle englobe ainsi que les espaces qui ont été prévus en dessous et entre celles-ci.

ALGO9 – Calculer la hauteur d'une Bounding Box

Données : Un graphe.

Résultat : /

Utilise : hauteur – un entier permettant de mémoriser la hauteur de la *Bounding Box* au fur et à mesure des calculs. Cet élément est initialisé à 0.

h – un entier permettant de calculer la hauteur d'une *Bounding Box*.

compteur – compteur permettant de retenir le nombre de suivants d'un nœud ayant déjà été visité.

couloirEntreBBox – couloir qui est toujours ajouté entre deux *Bounding Boxes*.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    h ← 0
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        Suivants ← suivants(nœud)
        Si (taille(Suivants) = 0)
            h ← hauteurBoîte(nœud)
        Sinon
        Début
            compteur ← 1
            j ← 0
            Tant que j < taille(Suivants)
            Début
                arc ← Suivants[j]
                VisiteArc(arc)
                h ← h + hauteur
                Si (compteur <> taille(Suivants) - 1)
                    h ← h + couloirEntreBBox
                compteur ← compteur + 1
                j ← j + 1
            Fin Tant que
        Fin Sinon
        h ← max(h, hauteurBoîte(nœud)
        hauteurBBox(nœud) ← h
        h ← h + (couloirsSousBBox(nœud) * largeurCouloir)
    Fin Si
    hauteur ← h
Fin Procédure
```

- LARGEUR DES *BOUNDING BOXES*

La largeur d'une *Bounding Box* se calcule en déterminant la largeur maximale parmi les *Bounding Boxes* qu'elle englobe, et en y ajoutant la somme des couloirs prévus avant et après cette Bounding Box.

Une séquence supplémentaire est prévue pour retenir la largeur de chacun des suivants d'un nœud, puisque la largeur d'une *Bounding Box* correspondant à un nœud est déterminée en fonction des largeurs des *Bounding Boxes* des suivants de ce nœud.

ALGO10 – Calculer la largeur d'une Bounding Box

Données : Un graphe.

Résultat : /

Utilise : largeur – un entier permettant de mémoriser la largeur de la *Bounding Box* au fur et à mesure des calculs. Cet élément est initialisé à 0.

l – un entier permettant de calculer la largeur d'une Bounding Box.

LargeurSuivants – une séquence permettant de retenir la largeur de chaque suivant d'un nœud. La taille de cette séquence varie selon le nombre de suivants du nœud. Ses éléments sont tous initialisé à 0.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    l ← 0
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        Suivants ← suivants(nœud)
        Si (taille(Suivants)<> 0)
        Début
            compteur ← 0
            j ← 0
            Tant que j < taille(Suivants)
            Début
                arc ← Suivants[j]
                VisiteArc(arc)
                LargeurSuivants[compteur] ← largeur
                largeur ← 0
                compteur ← compteur + 1
                j ← j + 1
            Fin Tant que
        Fin Si
        compteur ← 0
        j ← 0
        Tant que j < taille(Suivants)
        Début
            l ← max(l, LargeurSuivants[compteur]
            compteur ← compteur + 1
            j ← j + 1
        Fin Tant que
    Fin Si
    l ← l + largeurBoîte(nœud)
                + ( (maxAvant(nœud) + (maxAprès(nœud)) * largeurCouloir) )
    largeurBBox(nœud) ← l
    l ← l + couloirEntreBBox
    largeur ← l
Fin Procédure
```

G. Déterminer les coordonnées d'une Bounding Box

Avant de calculer le point d'encrage des boîtes, nous allons déterminer les coordonnées des *Bounding Boxes*. En effet, même si elles ne sont pas dessinées, les *Bounding Boxes* sont des rectangles qui ont une position relative sur le plan de dessin déterminé par les axes du système de coordonnées.

Pour assigner à chaque *Bounding Box* ses coordonnées, il suffit de parcourir le graphe et, lors de la visite de chaque nœud, de déterminer la position de la *Bounding Box* qui lui est associée. La première *Bounding Box* se voit assigner la coordonnée (0,0).

Pour déterminer les coordonnées des *Bounding Boxes* associées à chacun des nœuds du graphe, nous avons besoin des éléments calculés lors des précédentes étapes, comme les couloirs ajoutés, la hauteur des boîtes et les dimensions des *Bounding Boxes*.

Avant de passer au nœud suivant, si le nœud a des suivants, la coordonnée en X est augmentée de la somme de la largeur de la boîte avec les deux nombres maximaux de couloirs à ajouter avant et après la boîte correspondant au nœud. Cette modification assure le fait que la *Bounding Box* du nœud suivant soit située à bonne distance de celle du nœud en cours de traitement.

Par contre, si le nœud n'a pas ou plus de suivants, c'est-à-dire qu'il s'agit d'une feuille d'un sous-arbre, c'est la coordonnée en Y qui est modifiée. En effet, dans ce cas, c'est la somme de la hauteur du nœud avec les couloirs à ajouter sous la *Bounding Box* qui lui est additionnée.

Les procédures qui sont modifiées *sont VisiteNœud* et *VisiteGraphe*. En effet, c'est la première qui est désignée pour le calcul des coordonnées. La seconde ne varie que par le fait qu'elle prévoit l'ajout d'un espace entre chacune des *Bounding Boxes* englobant un sous-arbre du graphe.

De plus, un élément permettant la mémorisation des coordonnées de la *Bounding Box* associée à un nœud est nécessaire puisque chaque coordonnée est fonction de la précédente.

ALGO11 – Calculer les coordonnées d'une Bounding Box

Données : Un graphe.

Résultat : /

Utilise : coordX et coordY – deux entiers permettant de mémoriser les coordonnées au fur et à mesure du parcours du graphe.

coordX' et coordY' – deux entiers permettant de garder une copie des coordonnées lors du traitement d'un nœud.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    ...
    i ← 0
    Tant que i < taille(Nœuds) - 1
    Début
        nœud ← Nœuds[i]
        identifiant ← identifiant(nœud)
        (coordX,coordY) ← coordBBox(nœud)
        Si (Visite[identifiant] = VRAI) & (identifiant <> 0)
            coordY ← coordY + couloirEntreBBox
        VisiteNœud (nœud)
        i ← i + 1
    Fin Tant que
Fin Procédure
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        coordBBox(nœud) ← (coordX,coordY)
        Suivants ← suivants(nœud)
        Si taille(Suivants) = 0
            coordY ← coordY + hauteurBBox(nœud)
                            + (couloirsSousBBox(nœud) * largeurCouloir)
        Sinon
        Début
            (coordX',coordY') ← (coordX,coordY)
            coordX ← coordX + largeurBoîte(nœud) + couloirEntreBBox +
                    + ( (maxAvant(nœud) + (maxAprès(nœud)) * largeurCouloir) )
            j ← 0
            Tant que j < taille(Suivants)
            Début
                arc ← Suivants[j]
                VisiteArc(arc)
                Si (j < taille(Suivants) - 1)
                    coordY ← coordY + couloirEntreBBox
                j ← j + 1
            Fin Tant que
            coordY ← coordY' + hauteurBBox(nœud)
                            + (couloirsSousBBox(nœud) * largeurCouloir)
            (coordX, coordY) ← (coordX', coordY)
        Fin Sinon
    Fin Si
Fin Procédure
```

H. Déterminer le point d'encrage de chaque boîte

Après avoir calculer les coordonnées de la *Bounding Box* associée à chaque nœud du graphe, il reste encore à déterminer le point d'encrage de la boîte représentant graphiquement ces nœuds. En effet, les points d'encrage des boîtes ne sont pas identiques aux coordonnées des *Bounding Boxes*.

Nous devons tout d'abord tenir compte du nombre maximal de couloirs à ajouter devant la boîte pour calculer sa coordonnée en X.

Exemple 2.17

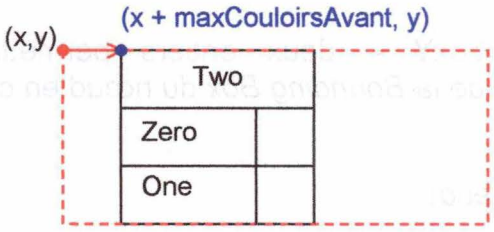


Figure 2.25 : Coordonnée en X d'une boîte.

Ensuite, pour avoir un dessin plus agréable, nous avons décidé que chaque boîte serait placée à mi-hauteur par rapport à la disposition de ses suivants. La coordonnée en Y d'une boîte est donc calculée en fonction du nombre de suivants du nœud correspondant à cette boîte, en ne considérant que les suivants appartenant au sous-arbre déterminé par le parcours du graphe.

Si un nœud a un nombre pair de suivants, le milieu de la boîte représentant ce nœud doit se situer au niveau du point situé au milieu des boîtes représentant la séquence de ses suivants. Le segment bleu de l'exemple 2.18 permet de visualiser cette décision.

Si, par contre, le nombre de suivants est impair, le milieu de la boîte doit se trouver à la même hauteur que le milieu de la boîte représentant le nœud au centre de la séquence des suivants. Dans ce cas, c'est le segment rouge de l'exemple 2.18 qui permet de le visualiser.

Exemple 2.18

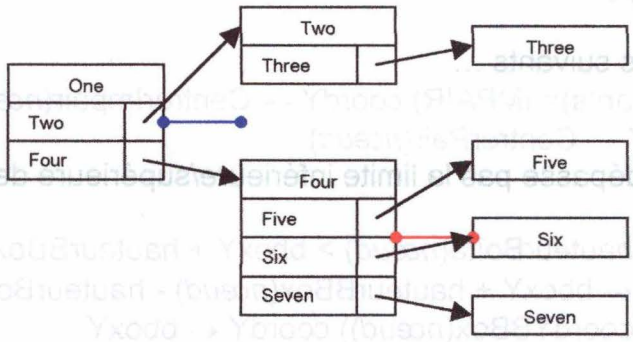


Figure 2.26 : Position relative des boîtes.

Il faudra cependant vérifier que la boîte n'est pas positionnée en dehors des limites fixées par la *Bounding Box* qui correspond au nœud qu'elle représente.

Pour déterminer les points d'encrage des boîtes représentant les nœuds du graphe, seule la procédure *VisiteNœud* de l'algorithme de parcours est modifiée. C'est en effet lors de la visite d'un nœud que la position de sa boîte est déterminée.

Deux procédures sont ajoutées afin de déterminer la coordonnée en Y de la boîte, l'une dans le cas où le nombre de suivants du nœud est pair, et l'autre dans le cas où ce nombre est impair.

ALGO12 – Calculer les coordonnées d'une boîte

Données : Un graphe.

Résultat : /

Utilise : bboxX et bboxY – deux entiers permettant de récupérer les coordonnées de la *Bounding Box* du nœud en cours de traitement.

Algorithme :

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (Visite[identifiant] = VRAI)
    Début
        Visite[identifiant] ← FAUX
        (bboxX,bboxY) ← coordBBox(nœud)
        coordY ← 0
        Suivants ← suivants(nœud)
        Si taille(Suivants) = 0
            coordY ← bboxY
        Sinon
        Début
            j ← 0
            Tant que j < taille(Suivants)
            Début
                arc ← Suivants[j]
                VisiteArc(arc)
                j ← j + 1
            Fin Tant que
        Fin Sinon
// Centrer la boîte sur les suivants ...
        Si (taille(Suivants)= IMPAIR) coordY ← CentrerImpair(nœud)
        Sinon coordY ← CentrerPair(nœud)
// Vérifier si la boîte ne dépasse pas la limite inférieure/supérieure de la Bounding
// Box
        Si (coordY + hauteurBoîte(nœud) > bboxY + hauteurBBox(nœud))
            coordY ← bboxY + hauteurBBox(nœud) - hauteurBoîte(nœud)
        Si (coordY < coordYBBox(nœud)) coordY ← bboxY

        coordBoîte(nœud) ← (bboxX + (maxAvant(nœud) * largeurCouloir),
                                                        coordY)
    Fin Si
Fin Procédure
```

```
Fonction coordY CentrerImpair(nœud)
Début
    Suivants ← suivants(nœud)
    arc ← TrouverArcArbreMilieu(Suivants)
    destination ← destination(arc)
    (coordX,coodY) ← coordBoîte(destination)
    coordY ← coordY + (hauteurBoîte(destination) / 2) - (hauteurBoîte(noeud) / 2)
    Renvoyer coordY
Fin Procédure

Fonction coordY CentrerPair(nœud)
Début
    Suivants ← suivants(nœud)
    arc ← TrouverArcArbreMilieu(Suivants)
    destination ← destination(arc)
    (coordX,coodY) ← coordBoîte(destination)
    coordY ← coodY + hauteurBoîte(destination) + (couloirEntreBBox / 2)
                                                - (hauteurBoîte(noeud) / 2)
    Renvoyer coordY
Fin Procédure
Fonction arc TrouverArcArbreMilieu(Suivants)
Début
    nbrArcArbre ← 0
    j ← 0
    Tant que j < taille(Suivants)
    Début
        arc ← Suivants[j]
        Si (type(arc) = FORWARD)
            nbrArcArbre ← nbrArcArbre + 1
        j ← j + 1
    Fin Tant que
    Si (taille(Suivants) = IMPAIR)
        compteur ← ArrondiInférieur(nbrArcArbre / 2)
    Si (taille(Suivants) = PAIR)
        compteur ← (nbrArcArbre / 2) - 1
    Fin Si
    j ← 0
    i ← 0
    Tant que (i <= compteur) & (j < taille(Suivants))
    Début
        Si (type(Suivants[j] = FORWARD)
            i ← i + 1
        j ← j + 1
    Fin Tant que
    Renvoyer Suivants[j-1]
Fin Procédure
```

I. Déterminer les points de routage de chaque arc

Les points de routage d'un arc sont les points par lesquels nous allons faire passer les arcs de manière à contourner les boîtes composant le dessin du graphe. Il s'agit donc de calculer les coordonnées de ces points.

De manière générale, une boîte est composée de compartiments. Le premier est destiné à contenir le type de l'objet représenté, et les éventuels autres compartiments sont chacun destiné à héberger un attribut de l'objet. Chacun de ces compartiments se termine par une case au centre de laquelle sera situé le point d'origine de l'arc issu de ce compartiment.

Exemple 2.19

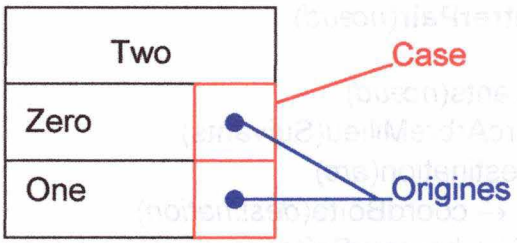


Figure 2.27 : Origine d'un arc.

Le point de destination d'un arc est, quant à lui, situé sur le côté gauche de la boîte qui correspond à la destination de l'arc. Il est en effet à mi-hauteur du premier compartiment de cette boîte.

Exemple 2.20

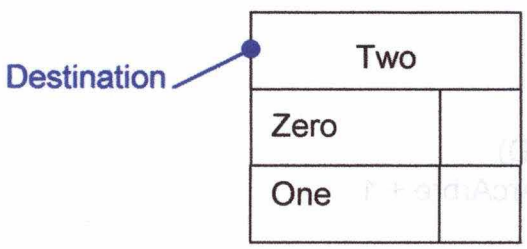


Figure 2.28 : Destination d'un arc.

Nous connaissons maintenant la manière de déterminer le point d'origine et celui de destination d'un arc, quelle que soit la catégorie à laquelle il appartient. Cependant, les points de routage ne sont pas toujours au nombre de deux. Nous allons donc examiner les différentes catégories d'arcs afin d'en déterminer les caractéristiques particulières liées au routage des arcs.

Comme nous l'avons vu auparavant, il y a deux catégories d'arcs : les arcs d'arbre et les arcs auxiliaires.

Nous nous sommes d'abord intéressés aux arcs d'arbre, ceux-ci n'étant composés que d'un seul segment. En effet, un arc d'arbre est représenté par une simple droite reliant le nœud d'origine et le nœud de destination. D'où, pour chacun des arcs d'arbre du graphe, seul deux points de routage sont nécessaires : le point d'origine et le point de destination, pour lesquels nous avons déjà défini la position. L'exemple 2.18 permet au lecteur de se rendre compte de la manière dont sont dessinés ces arcs.

Ensuite, nous nous sommes occupés des arcs auxiliaires qui sont soit des arcs arrières, soit des arcs transversaux. Dans les deux cas, l'arc passera par six points de routage (dont l'origine et la destination). Les points de routage sont numérotés de 0 à 6, 0 étant le point d'origine et 6 le point de destination. Les coordonnées des points d'origine et de destination de ces arcs sont déterminées de la même manière que pour les arcs d'arbre. Les coordonnées des quatre points intermédiaires sont encore à déterminer. Nous rappelons au lecteur que l'exemple 2.6 présente une illustration de la manière dont sont disposés les points de routage des arcs *auxiliaires*.

La position des points de routage d'un arc est déterminée en fonction des couloirs prévus pour le passage de celui-ci. En effet, comme nous l'avons expliqué dans le module concernant l'ajout des couloirs, à chaque nœud est associé un attribut permettant de connaître le nombre maximal de couloirs à ajouter avant et après la boîte qui le représente, ainsi que le nombre de couloirs à ajouter sous la *Bounding Box* qui lui est associée.

De plus, nous avons vu que l'assignation des arcs aux couloirs se fait selon certaines règles. En effet, les arcs sont dessinés à une certaine distance de la boîte d'origine selon qu'ils sont issus du premier ou du dernier compartiment de cette boîte. Il en est de même pour la distance séparant l'arc de sa boîte de destination. Pour respecter ces décisions, il faut choisir, pour chaque point de routage, le couloir adéquat.

Ces règles ne sont en fait que le résultat de décisions arbitraires qui ont été prises lors d'un premier essai de routage des arcs. Il est évident que notre première tentative, qui est l'objet des explications à venir, n'est pas la solution la plus favorable à la réalisation du facteur esthétique qui fait partie des paramètres de dessin de notre utilitaire. Ce point sera d'ailleurs débattu au sein du chapitre 4.

Pour en revenir à l'algorithme de routage actuel, les calculs concernant les points de routage des arcs arrières et ceux des arcs transversaux diffèrent par le fait que les arcs arrières passent sous la *Bounding Box* du nœud de destination, tandis que les arcs transversaux passent sous un autre nœud, choisi selon la méthode décrite précédemment.

Les trois types d'arcs sont donc routés de manières différentes, d'où une procédure appropriée est associée à chacun de ceux-ci. C'est la procédure *VisiteArc* de l'algorithme de parcours du graphe qui est chargée de faire appel à la procédure correspondant au type d'arc visité. Cependant, comme les procédures concernant les arcs auxiliaires ne diffèrent qu'en un point, le calcul du point de routage 2, seule la première procédure sera décrite en entier.

Afin de sélectionner les couloirs devant contenir un arc, trois séquences sont nécessaires. La première est composée d'autant d'éléments qu'il y a de nœuds dans le graphe et permet de conserver, pour chacun d'eux, le nombre de couloirs qui ne sont pas encore occupés devant leur boîte. La seconde contient le numéro du couloir qui sera utilisé pour le prochain arc à faire passer sous la *Bounding Box* d'un nœud. La troisième est composée d'autant d'éléments qu'il y a de niveaux de dessin et sert à mémoriser pour chacun d'eux le nombre de couloirs qui ne sont pas encore occupés par un arc.

La procédure *VisiteGraphe* est, elle aussi, modifiée par l'appel d'une procédure ayant pour objectif d'initialiser ces séquences.

ALGO13 – Calculer les points de routage des arcs

Données : Un graphe, le nombre de niveaux de dessin du graphe.

Résultat : /

Utilise : noSuivant – entier associé à chaque arc permettant de connaître la position qu'il occupe dans la séquence des suivants du nœud dont il est issu.

nbrAvant – attribut d'un nœud contenant le nombre de couloirs qui ne sont pas encore occupés avant une boîte.

nbrAprès – attribut d'un nœud contenant le nombre de couloirs qui ne sont pas encore occupés après le niveau de la boîte.

nbrSous – attribut d'un nœud contenant le numéro du couloir qui doit être occupé par le prochain arc passant sous la Bounding Box.

pointRoutageO – coordonnées (coordX,coordY) du point de routage correspondant à l'origine de l'arc.

pointRoutage1 – coordonnées du deuxième point de routage.

pointRoutage2 – coordonnées du troisième point de routage d'un arc.

pointRoutage3 – coordonnées du quatrième point de routage d'un arc.

pointRoutage4 – coordonnées du cinquième point de routage d'un arc.

pointRoutageD – coordonnées du point de routage correspondant à la destination de l'arc.

EspacePointBordBoîte – entier correspondant à la distance fixée entre un point d'origine et le bord de la boîte à laquelle il appartient.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    InitialiserSéquences(graphe)
    ...
Fin Procédure

Procédure InitialiserSéquences(graphe)
Début
    Nœuds ← nœuds(graphe)
    i ← 0
    Tant que i < taille(Nœuds)
    Début
        nœud ← Nœuds[i]
        nbrAvant(nœud) ← couloirsAvant(nœud)
        niveau ← niveau(nœud)
        nbrAprès(niveau) ← maxAprès(nœud)
        nbrSous(nœud) ← 1
        i ← i + 1
    Fin Tant que
Fin Procédure
```

```
Procédure VisiteArc (arc)
Début
    destination ← destination(arc)
    VisitNœud(destination)
    SI (type(arc) = FORWARD)
        CalculerPointsRoutageArcArbre(arc)
    SI (type(arc) = BACK)
        CalculerPointsRoutageArcArrière(arc)
    SI (type(arc) = CROSS)
        CalculerPointsRoutageArcTransversal(arc)
    Fin Si
Fin Procédure

Fonction point CalculerPointOrigine(arc)
Début
    origine ← origine(arc)
    (coordX,coordY) ← coordBoîte(origine)
    coordX ← coordX + largeurBoîte(origine) - EspacePointBordBoîte
    coordY ← coordY + (noSuivant(arc) * hauteurCompartiment) +
                                                (hauteurCompartiment/2)
    Renvoyer (coordX,coordY)
Fin Procédure

Fonction point CalculerPointDestination(arc)
Début
    destination ← destination(arc)
    (coordX,coordY) ← coordBoîte(origine)
    coordY ← coordY + (hauteurCompartiment/2)
    Renvoyer (coordX,coordY)
Fin Procédure

Procédure CalculerPointsRoutageArcArbre(arc)
Début
    pointRoutageO(arc) ← CalculerPointOrigine(arc)
    pointRoutageD(arc) ← CalculerPointDestination(arc)
Fin Procédure

Fonction nœud TrouverNoeud(arc)
Début
    Nœuds ← nœuds(graphe)
    i ← 0
    Tant que (i < taille(Nœuds)) & (ArcSous[arc, i] <> VRAI)
        i ← i + 1
    Si (ArcSous[arc, i] = VRAI)
        Renvoyer Nœuds[i]
Fin Procédure
```

```
Procédure CalculerPointsRoutageArcArrière (arc)
Début
    pointRoutageO(arc) ← CalculerPointOrigine(arc)
    pointRoutageD(arc) ← CalculerPointDestination(arc)
    (origX,origY) ← pointRoutageO(arc)
    (destX,destY) ← pointRoutageD(arc)
    (coordX,coordY) ← (origX,origY)
// point de routage 1
    niveau ← niveau(origine(arc))
    coordX    ← origX + EspacePointBordBoîte +
                        (largeurCouloir * nbrAprès(niveau)) - (largeurCouloir / 2)
    nbrAprès(niveau) ← nbrAprès(niveau) - 1
    pointRoutage1(arc) ← (coordX,coordY)
// point de routage 2
    nœud ← TrouverNoeud(arc) // nœud sous lequel passe l'arc
    (bboxX,bboxY) ← coordBBox(nœud)
    coordY ← bboxY + hauteurBBox(nœud) +
                        (largeurCouloir  * nbrSous(nœud)) - (largeurCouloir / 2)
    nbrSous(nœud) ← nbrSous(nœud) + 1
    pointRoutage2(arc) ← (coordX,coordY)
// point de routage 3
    destination ← destination(arc)
    (bboxX,bboxY) ←coordBBox(destination)
    coordX ← bboxX + (largeurCouloir * nbrAvant(destination)) - (largeurCouloir / 2)
    nbrAvant(destination) ← nbrAvant(destination) - 1
    pointRoutage3(arc) ← (coordX,coordY)
// point de routage 4
    coordY ← destY
    pointRoutage4(arc) ← (coordX,coordY)
Fin Procédure
```

```
Procédure CalculerPointsRoutageArcTransversal (arc)
Début
    ...
// point de routage 2
    destination ← destination(arc)
    (bboxX,bboxY) ← coordBBox(destination)
    coordY ← bboxY + hauteurBBox(destination) +
                    (largeurCouloir * nbrSous(destination)) - (largeurCouloir / 2)
    nbrSous(destination) ← nbrSous(destination) + 1
    pointRoutage2(arc) ← (coordX,coordY)
// point de routage 3
    coordX ← bboxX + (largeurCouloir * nbrAvant(destination)) - (largeurCouloir / 2)
    nbrAvant(destination) ← nbrAvant(destination) - 1
    pointRoutage3(arc) ← (coordX,coordY)
    ...
Fin Procédure
```

J. Dessiner le graphe

Jusqu'ici, nous avons présenté tous les modules de l'utilitaire qui sont consacrés au calcul des divers éléments nécessaires à déterminer les positions respectives des boîtes et des arcs. Chaque boîte connaît dès à présent les coordonnées de son point d'encrage sur le plan de dessin, et chaque arc connaît celles des points par lesquels il doit passer. Ces informations vont donc enfin nous permettre de dessiner le graphe.

Pour dessiner le graphe, il suffit de le parcourir et de faire appel, pour chaque nœud et pour chaque arc visité, à la procédure qui permet de le dessiner. L'algorithme de *parcours du graphe* n'est dans ce cas modifié en rien mis à part le fait que *VisiteNœud* appelle la procédure de dessin d'un nœud et que *VisiteArc* appelle celle de dessin d'un arc.

La description de ces procédures nécessite, au préalable, quelques explications au sujet des procédures de dessin. En effet, en Java, des méthodes sont prévues pour dessiner un point, une ligne, un rectangle et toutes sortes de formes, ainsi que des caractères et d'autres objets.

Cependant, comme l'objectif de ce dernier algorithme n'est pas de s'attarder aux éléments propres à Java, nous allons utiliser des procédures qui permettent de dessiner les éléments graphiques dont nous avons besoin. Ces éléments sont au nombre de trois, l'un permettant de dessiner une ligne entre deux points dont les coordonnées sont connues, l'autre permettant de dessiner une boîte en supposant que les compartiments sont également dessinés, ainsi que les attributs, et le dernier permettant de dessiner une flèche.

Ces procédures sont donc :

- DessinerLigne(*origine*, *destination*)
- DessinerBoîte(*coordonnées*, *largeur*, *hauteur*)
- DessinerFlêche(*coordonnées*)

ALGO14 – Dessiner le graphe

Données : Un graphe.

Résultat : /

Utilise : /

Algorithme :

```
Procédure VisiteNœud(noeud)
Début
    ...
    DessinerBoîte(coordBoîte(nœud), largeurBoîte(nœud), hauteurBoîte(nœud))
Fin Procédure

Procédure VisiteArc (arc)
Début
    ...
    DessinerArc(arc)
Fin Procédure

Procédure DessinerArc (arc)
Début
    Si (type(arc) = FORWARD)
    Début
        DessinerLigne(pointRoutageO(arc), PointRoutageD(arc))
        DessinerFlêche(PointRoutageD(arc))
    Fin Si
    Sinon
    Début
        DessinerLigne(pointRoutageO(arc), PointRoutage1(arc))
        DessinerLigne(pointRoutage1(arc), PointRoutage2(arc))
        DessinerLigne(pointRoutage2(arc), PointRoutage3(arc))
        DessinerLigne(pointRoutage3(arc), PointRoutage4(arc))
        DessinerLigne(pointRoutage4(arc), PointRoutageD(arc))
        DessinerFlêche(PointRoutageD(arc))
    Fin Sinon
Fin Procédure
```

2.5 OPTIMISATIONS

Les graphes représentant l'environnement et le store d'un programme Java peuvent parfois atteindre une profondeur insoupçonnée. Ce faisant, l'utilisateur ne désire pas toujours obtenir toutes les informations d'un coup. Il peut ne s'intéresser qu'à une partie du graphe en particulier. Pour lui permettre de ne visualiser que ce qu'il désire observer, l'utilitaire est pourvu de plusieurs éléments dont une limitation en profondeur de l'affichage du graphe et un zoom.

En plus de cela, quelques propriétés supplémentaires ont été ajoutées afin de rendre le dessin plus lisible selon la sensibilité de l'utilisateur.

2.5.1 *Limiter la profondeur du dessin*

Limiter la profondeur du dessin signifie que l'utilisateur peut demander que seuls les nœuds dont le niveau de dessin est inférieur ou égal à la profondeur désirée soient affichés.

Il est facile de demander, via l'interface graphique, la profondeur jusqu'à laquelle l'utilisateur désire afficher le graphe. Comme un niveau de dessin est associé à chaque nœud du graphe au début des étapes précédant le dessin du graphe, il reste à ne traiter, lors des calculs effectués au cours de ces étapes, que les nœuds dont le niveau de dessin est inférieur à la profondeur entrée.

Il nous faut donc ajouter les conditions liées à la profondeur demandée dans tous les algorithmes de calcul et de dessin. Pour ce faire, nous avons utilisé une séquence de booléens, contenant un élément par nœud du graphe, dont la valeur est à VRAI si le nœud doit être traité et à faux dans le cas contraire. Or cette séquence n'est autre que la séquence Visite de l'algorithme de parcours du graphe initialisée en fonction de la profondeur demandée et du niveau de dessin de chaque nœud. La valeur de Visite des nœuds ayant un niveau de dessin supérieur à la profondeur demandée est mise à FAUX.

Cette séquence, une fois initialisée par l'algorithme ALGO15, est passée en argument à tous les autres algorithmes de calculs précédant le dessin du graphe de manière à ce que seuls les nœuds dont la valeur de Visite est à VRAI soient traités.

Il se peut que certaines conditions supplémentaires soient nécessaires dans ces algorithmes, mais, ne pouvant décrire à nouveau tous ces algorithmes, nous n'entrerons pas dans les détails de ces modifications.

En ce qui concerne le dessin à proprement dit, les arcs issus du dernier niveau demandé se terminent par des pointillés pour montrer que l'arbre n'est pas dessiné dans son entièreté.

Exemple 2.21

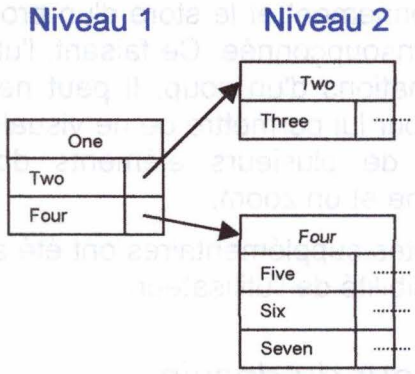


Figure 2.29 : Limite de la profondeur du dessin à 2.

ALGO15 – Limiter la profondeur du dessin

Données : Un graphe, la profondeur demandée.

Résultat : /

Utilise : Visite[] – une séquence de booléens permettant de savoir si un nœud doit encore être visité ou pas. Elle contient autant d'éléments que de nœuds dans le graphe. Chacun de ces éléments est initialisé à VRAI. Cette séquence tient compte de la profondeur demandée et sera utilisée dans les autres algorithmes.

VisiteDfs[] – une séquence de booléens permettant de savoir si un nœud doit encore être visité ou pas. Elle contient autant d'éléments que de nœuds dans le graphe. Chacun de ces éléments est initialisé à VRAI.

compteur – entier permettant de déterminer la profondeur du nœud visité. Il est initialisé à 0.

Algorithme :

```
Procédure VisiteGraphe (graphe)
Début
    Nœuds ← nœuds(graphe)
    i ← 0
    Tant que i < taille(Nœuds)
    Début
        VisiteNœud(Nœuds[i])
        i ← i + 1
    Fin Tant que
Fin Procédure
```

```
Procédure VisiteNœud(nœud)
Début
    identifiant ← identifiant(nœud)
    Si (VisiteDfs[identifiant] = VRAI) & (Visite[identifiant] = VRAI)
    Début
        Si (compteur > profondeur)
            Visite[identifiant] ← FAUX
        VisiteDfs[identifiant] ← FAUX
        Suivants ← suivants(nœud)
        j ← 0
        Tant que j < taille(Suivants)
        Début
            compteur ← compteur + 1
            arc ← Suivants[j]
            VisiteArc(arc)
            compteur ← compteur - 1
            j ← j + 1
        Fin Tant que
    Fin Si
Fin Procédure
```

2.5.2 Zoom

En plus de limiter la profondeur de dessin du graphe, il semble utile de pourvoir l'utilitaire de zooms. Deux possibilités de zooms s'offrent à nous, la première permettant de faire un zoom sur l'entièreté du dessin et la seconde se limitant à une partie du dessin préalablement sélectionnée par l'utilisateur.

A l'heure actuelle, seule la première solution est envisagée.

Dans un premier temps, lors du premier affichage du graphe, l'utilitaire effectue un zoom (positif ou négatif) afin de faire entrer la totalité du dessin dans le plan de dessin visible à l'écran. Ensuite, l'utilisateur peut choisir, par l'intermédiaire de deux glissières, le pourcentage de zoom qu'il désire voir appliquer au dessin.

Plus d'explications sur la manière dont les zooms peuvent être réalisés en Java sont apportées dans le chapitre 3.

2.5.3 Autres propriétés

D'autres propriétés ont été ajoutées au dessin de manière à rendre celui-ci plus agréable et plus lisible. Ces propriétés sont en fait laissées à l'appréciation de l'utilisateur. Il peut donc choisir les couleurs des arcs en fonction de leur type, il peut également décider que les arcs d'arbre soient routés de manière similaire aux autres arcs, il a également la possibilité de choisir un meilleur agencement des arcs afin de limiter un tant soit peu les croisements entre ceux-ci, il peut encore choisir de ne pas garder les proportions du dessin initial.

Nous ne rentrons pas dans les détails de ces aspects, ceux-ci n'étant pas encore au point.

CHAPITRE 3

LES ASPECTS INTÉRESSANTS DE JAVA ET L'IMPLÉMENTATION

L'utilitaire d'affichage a été réalisé en utilisant le langage de programmation Java, un langage orienté objet. Certains aspects de ce langage ont été particulièrement intéressants, qu'ils soient ou non propres à Java. Parmi ces éléments, on retrouve les éléments de Java qui permettent de dessiner et les itérateurs.

Ces aspects présentés, nous mettrons l'accent sur la méthode utilisée pour l'implémentation des algorithmes de calcul.

Ensuite, nous aborderons l'implémentation à proprement dite en décrivant brièvement les classes correspondant aux structures de données et l'architecture de l'application.

3.1 LE DESSIN EN JAVA

Un des aspects le plus intéressant de Java, dans le cadre de l'utilitaire d'affichage, est la possibilité que ce langage offre en termes de graphisme. En effet, il est aisé de dessiner des lignes, des formes, des images et du texte avec des polices, des styles et des couleurs différents. Nous pouvons également créer des applications qui réagissent à la souris ou au clavier. De plus, les outils permettant de faire des zooms, des rotations, des translations et autres sont fournis par Java.

Les classes qui permettent cela sont *Graphics* et *Graphics2D*. Avant d'en expliquer les principes, nous allons mettre le doigt sur la particularité du système de coordonnées de Java.

Dans ce chapitre, nous avons utilisé les documents suivants : *[JAVADOC]*, *[JAVATUT]*, *[JAVACORE]* et *[JAVAMAC]*.

3.1.1 Le système de coordonnées

Le système de coordonnées habituellement utilisé en dessin ressemble à celui de la figure 2.11. Cependant, le système de coordonnées de Java n'est pas identique à celui-là, bien qu'il en soit proche. En fait, la seule chose qui les différencie est que le système de Java est renversé. La figure 3.1 permet une meilleure visualisation de ce renversement.

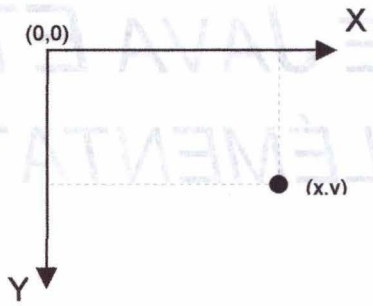


Figure 3.1 : Le système de coordonnées de Java.

3.1.2 Graphics

La plupart des fonctions graphiques de Java sont accessibles via une seule classe, la classe *Graphics*, qui est disponible dans le package AWT (*Abstract Windowing Toolkit*) de Java.

Un package regroupe des classes ayant un certain nombre de points communs. Ils permettent entre autres d'organiser le code source et de simuler l'héritage multiple.

Le package AWT est un des packages qui fait partie de l'API (*Application Programming Interfac*) de Java. Cette API a été développée par les concepteurs de Java. Les classes de l'API sont regroupées en package, chacun d'entre eux pouvant comporter plusieurs classes et interfaces.

Les interfaces sont les cousines des classes avec moins de privilèges. Les classes peuvent définir un objet ; les interfaces définissent un jeu de méthodes et de constantes qui seront implémentées par un objet.

La classe *Graphics* modélise un contexte graphique. Un contexte graphique est une représentation abstraite de la surface graphique sur laquelle nous allons dessiner. C'est essentiellement un moyen de nous permettre de faire appel aux routines de dessin sans avoir à nous inquiéter de la manière dont nous allons publier le dessin, soit en l'affichant à l'écran soit en l'imprimant sur une feuille.

Le résultat de l'application de fonctions graphiques sur un objet de type *Graphics* est toujours lié à un composant. Les composants Java sont modélisés au plus haut niveau de l'API Java par la classe *Component* dans le package AWT. C'est donc un élément générique qui est à la base de tous les autres objets graphiques du système Java. Il est possible de dessiner sur n'importe quel objet dérivé de la classe *Component*. En fait, tous les objets *Component* ont un objet de type *Graphics* qui leur est associé et qui est utilisé pour dessiner sur sa surface. Celui-ci peut cependant ne pas être utile, comme dans le cas d'une liste ou d'une *checkbox*.

La classe *Graphics* est une classe abstraite, qui ne peut donc pas être instanciée. En effet, travailler avec de tels objets demande une connaissance détaillée de la plate-forme sur laquelle le programme tourne car les fonctions de dessins sont réalisées par des classes concrètes qui sont fortement liées à une plate-forme particulière. C'est pour cette raison que la Machine Virtuelle de Java est fournie avec les classes concrètes nécessaires à l'environnement sur lequel elle sera implantée. Il n'y a donc pas lieu de s'inquiéter à propos des classes spécifiques à la plate-forme ; une fois que l'objet de type *Graphics* est obtenu, toutes les méthodes de la classe *Graphics* peuvent être appelées, les classes spécifiques à la plate-forme s'occupent d'exécuter le programme conformément à l'environnement.

Comme il est impossible de créer explicitement d'objet de type *Graphics*, la question de savoir comment obtenir un tel objet se pose. Il se fait que l'objet *Graphics* associé à un objet *Component* est transmis à sa méthode *paint* ou *update*. Les dessins sont donc généralement implémentés dans une méthode *paint* qui redéfinit la méthode du comosant. Il est également possible d'obtenir l'objet de type *Graphics* d'un composant en appelant la méthode *getGraphics* de la classe *Component*.

La plupart des opérations fournies par la classe *Graphics* ont un lien avec l'une des trois catégories suivantes : dessiner des éléments primitifs (ligne, forme ...), dessiner du texte ou dessiner des images.

Nous avons, dans un souci d'actualité, utilisé Swing. Swing est la librairie permettant de créer des interfaces graphiques dans la version 2 de Java. Elle semble être plus robuste et propose plus d'éléments graphiques que l'AWT. Elle est également plus portable et plus facile à utiliser que ce dernier.

Voici le schéma qui permet de situer les classes *JPanel* et *JFrame* de Swing par rapport à celle de l'AWT :

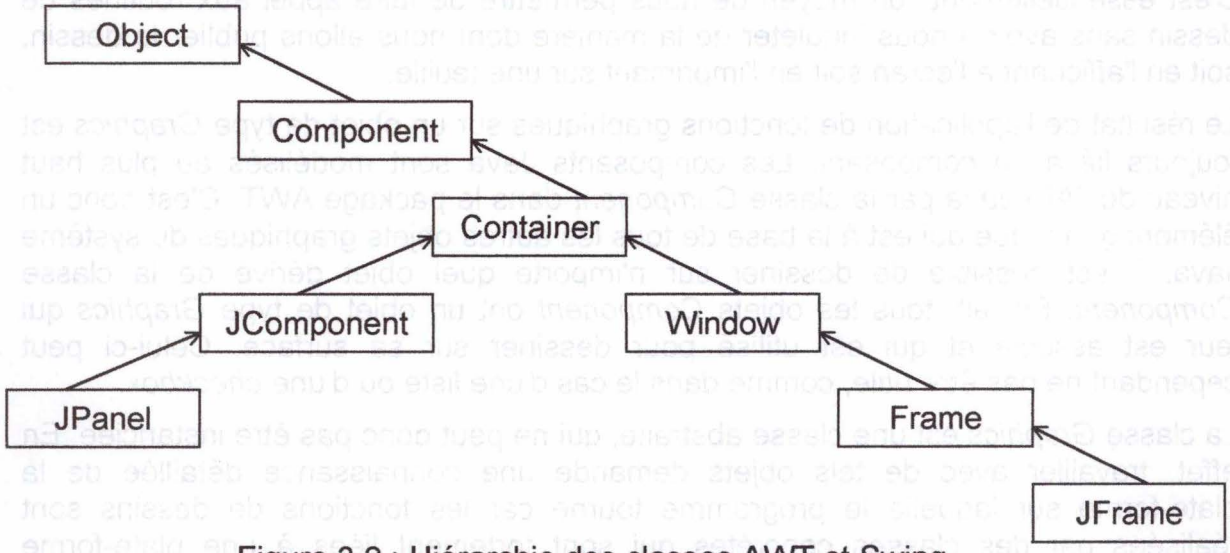


Figure 3.2 : Hiérarchie des classes AWT et Swing.

La technique de dessin dans Swing est similaire à celle de l'AWT mis à part qu'il s'agit de *JComponent* et que la méthode permettant le dessin est *paintComponent*.

Le composant le plus approprié comme plan de dessin est le *panel* (*JPanel*). Les *frames* (*JFrame*) sont quant à elles moins appropriées pour ce genre de choses, elles sont plutôt destinées à contenir des *panels*, des boutons et autres composants. La technique habituelle est donc d'insérer un *panel* dans une *frame*.

Pour pouvoir dessiner dans le *panel*, il suffit de créer une classe qui étend la classe *JPanel* et d'y redéfinir la méthode *paintComponent* puisque c'est elle qui reçoit le contexte graphique en argument comme nous l'avons vu pour la méthode *paint*.

Les méthodes de la classe *Graphics* sont entre autre *drawString*, *drawLine*, *drawRect*. Ce sont ces méthodes que nous avons utilisées pour dessiner le graphe.

Exemple 3.1

Cette méthode est la méthode qui permet de dessiner une boîte, c'est-à-dire la méthode *paint* de la classe *InstanceBox*.

```
public void paint(Graphics g, int depth, Properties properties) {
     g.setFont(_font);
     _anchor = _dec.getAnchor();
     g.setColor(Color.black);
     g.drawRect( (int)_anchor.getX(),(int)_anchor.getY(),
                 _widthBox, _heightBox);
     g.drawLine( (int)_anchor.getX() + _widthBox
                      - DrawingConstants.SPACE_BTWN_VLINE_EDGEBOX,
                 (int)_anchor.getY() + DrawingConstants.HEIGHT_FIELD,
                 (int)_anchor.getX() + _widthBox
                      - DrawingConstants.SPACE_BTWN_VLINE_EDGEBOX,
                 (int)_anchor.getY() + _heightBox);
     drawName(g);
     drawFields(g);
     drawArcs(g, depth, properties);
}
```

3.1.3 Graphic2D

La classe *Graphics2D* étend la classe *Graphics* et procure un contrôle plus sophistiqué aux niveaux de la géométrie, de la transformation des coordonnées, de la gestion des couleurs et de la mise en page du texte.

Toutes les coordonnées passées à un objet de la classe *Graphics2D* sont spécifiées dans un système de coordonnées indépendant du dispositif qui affichera le dessin (écran, imprimante ...). Ce système de coordonnées est appelé *User Space* et n'est utilisé que par les applications.

L'objet *Graphics2D* contient un objet *AffineTransform* qui représente une partie du rendu du dessin. Il défini comment convertir les coordonnées du *User Space* vers le système de coordonnées propre au dispositif affichant le dessin, le *Device Space*.

Pour effectuer des transformations supplémentaires, comme les rotations ou les zooms, il est possible d'ajouter des transformations au contexte de l'objet *Graphics2D*. Celles-ci font partie de l'ensemble des transformations appliquées à l'objet durant son dessin.

Les objets *AffineTransform* sont utilisés pour transformer du texte, des formes et des images au moment de leur dessin. Les éléments graphiques restent du même type après la transformation, c'est-à-dire que les lignes droites restent des lignes droites, et les lignes parallèles le sont toujours après la transformation, même si les distances entre les points et les angles entre les lignes non parallèles peuvent être modifiés.

Il s'agit en fait de transformations linéaires sur un ensemble de primitives graphiques. Une transformation peut donc être une séquence de translations, zooms, rotations et retournements.

De telles transformations de coordonnées peuvent être représentées par des matrices 3x3 avec une dernière colonne implicitement à [0 0 1]. Une matrice de ce type transforme les coordonnées (x, y) en coordonnées (x', y') en les considérant comme des vecteurs colonnes et en multipliant le vecteur de coordonnées par une matrice selon le procédé suivant :

$$[x'] = [m00\ m01\ m02][x] = [m00\,x + m01\,y + m02]$$

$$[y'] = [m10\ m11\ m12][y] = [m10\,x + m11\,y + m12]$$

$$[1] = [0\ \ 0\ \ 1]\ \ [1] = [1]$$

Les transformations peuvent être combinées, créant une série de transformations qui peuvent être appliquées à un objet. Cette combinaison est considérée comme la concaténation des transformations qui la constituent. Il est donc possible de concaténer des transformations entre elles en concaténant des objets *AffineTransform*. La dernière transformation ajoutée est la première à être appliquée.

Exemple 3.2

L'exemple suivant permet de voir l'une des utilisations de l'objet *Graphic2D* et de son *AffineTransform*. En effet, pour dessiner une flèche au bout d'un arc, cet arc étant oblique par rapport à l'axe X du système de coordonnées, et pour que la flèche ait la même inclinaison que l'arc, nous avons besoin de faire une rotation correspondant à l'angle formé par l'arc et la demi-droite parallèle à l'axe X qui part de l'origine de l'arc.

Pour ce faire, nous avons une méthode qui a pour tâche de dessiner un arc et sa flèche. Cette méthode fait partie de celles qui sont applicables à un objet de la classe *InstanceArc*.

```
private void drawArcAndArrow(Graphics g) {
// Dessin de l'arc
    int nbrRP = _routingPoints.length-1;
    int i = 0;
    while (i < nbrRP) {
        int j = i + 1;
        g.drawLine( (int)_routingPoints[i].getX(),
                    (int)_routingPoints[i].getY(),
                    (int)_routingPoints[j].getX(),
                    (int)_routingPoints[j].getY());
        i++;
    }
// Calcul de l'angle de l'arc
    Point coordOrig = _routingPoints[nbrRP - 1];
    Point coordDest = _routingPoints[nbrRP];
    double hypot = Math.sqrt( Math.pow(coordDest.getX() -
                                       coordOrig.getX(), 2.0) +
                              Math.pow(coordDest.getY()-
                                       coordOrig.getY(), 2.0));
    double height = coordDest.getY() - coordOrig.getY();
    double asin   = height/hypot;
    double angle  = Math.asin(asin);
// Dessin de la fleche
    Graphics2D g2 = (Graphics2D) g;
    ArrowIcon arrow = new ArrowIcon(SwingConstants.RIGHT, coordDest);
    AffineTransform save = g2.getTransform();
    AffineTransform trans = new AffineTransform();
    trans.rotate(angle, (int)coordDest.getX(), (int)coordDest.getY());
    g2.transform(trans);
    arrow.paint(g2);
    g2.setTransform(save);
}
```

Nous ferons remarquer au lecteur que, puisque seule la flèche doit subir une rotation, *l'AffineTransform*, avant d'être modifié pour le dessin de la flèche, doit être sauvé pour être rétabli après le dessin de celle-ci.

3.2 LES ITÉRATEURS

Dans les algorithmes exposés ci-dessus, nous avons fréquemment dû parcourir des séquences d'éléments. Il existe une manière de masquer le fait que ce parcours nécessite un compteur qui est incrémenté au fur et à mesure. En effet, les API de Java comportent des interfaces dont les méthodes sont destinées à permettre d'effectuer les opérations couramment utilisées sur une séquence : vérifier qu'il y a encore un élément dans la séquence, obtenir l'élément suivant ...

L'une de ces interfaces est *Iterator*. Les méthodes prévues par cette interface sont au nombre de trois :

- *hasNext* qui renvoie VRAI si l'itération est possible, c'est-à-dire s'il reste encore un élément dans la collection ;
- *next* qui renvoie l'élément suivant ;
- *remove* qui enlève le dernier élément de la collection renvoyé par l'itérateur.

Cependant, comme nous avions besoin d'un itérateur qui permette le parcours d'une séquence dans les deux sens, nous avons décidé de créer notre propre interface.

Afin de parcourir les séquences de nœuds, d'arcs ou d'autres séquences, nous utilisons l'itérateur décrit par l'interface *GIterator*. Notre interface se compose donc des méthodes suivantes :

- *next* qui permet de passer à l'élément suivant ;
- *previous* qui permet de passer à l'élément précédent ;
- *getCurrent* qui permet d'obtenir l'élément courant ;
- *hasMore* qui détermine s'il y a un élément suivant l'élément courant ;
- *hasLess* qui détermine s'il y a un élément précédent l'élément courant ;
- *reset* qui remet l'itérateur au début de la séquence ;
- *last* qui remet l'itérateur en fin de séquence ;
- *add* qui permet d'ajouter un élément à la séquence après l'élément courant.

L'interface *GIterator* est implémentée par trois classes :

- ***GIteratorV*** pour les séquences de *Vertices* ;
- ***GIteratorA*** pour les séquences de *Arcs* ;
- ***GIteratorIA*** pour les séquences de *InstanceArc*.

Ces classes sont quasiment identiques, à l'exception des types d'objets sur lesquels les itérateurs sont appliqués.

Pour avoir un meilleur aperçu de l'utilité de *GIterator*, nous présentons ci-après le code Java qui a pour objectif de permettre l'impression de chacun des nœuds du graphe. Il s'agit en fait d'un simple parcours du graphe.

Exemple 3.3 – Parcours du graphe avec GIterator

```
package Function;

import GUI.*;
import Graph.*;
import GIterator.*;

public class FDfsPrint implements Function {
    private boolean[] _visitDfs;

    public FDfsPrint(int nbrVertices) {
        _visitDfs = new boolean[nbrVertices];
        for (int i = 0; i < nbrVertices; i++) _visitDfs[i] = false;
    }
    public void applyGraph(Graph graph) {
        GIterator it = graph.getVertices();
        it.reset();
        while (it.hasMore()) {
            Vertex v = (Vertex)it.getCurrent();
            v.apply(this);
            it.next();
        }
    }
    public void applyArc(Arc arc) {
        arc.getSink().apply(this);
    }
    public void applyVertex(Vertex vertex){
        if(_visitDfs[vertex.getId()] == false) {
            _visitDfs[vertex.getId()] = true;
            System.out.println(vertex.toString());
            GIterator it = vertex.getDescendents();
            while(it.hasMore()) {
                Arc arc = (ArcI)it.getCurrent();
                arc.apply(this);
                it.next();
            }
        }
    }
}
```

3.3 LE PACKAGE FUNCTION

Lors de la réalisation de l'utilitaire d'affichage, un aspect propre à la programmation orientée objet nous a particulièrement intéressé de par sa subtilité. En effet, comme nous l'avons vu dans le chapitre précédent, tous les algorithmes de calcul utilisés par l'utilitaire d'affichage de l'environnement et du store ont la même structure. Nous avons alors décidé de faire un package regroupant tous ces calculs et de lui associer une interface.

Nous avons donc un package *Function* qui se compose d'une interface et de toutes les classes qui implémentent les algorithmes définis ci-dessus. L'interface est également appelée *Function* et détermine la structure que toutes les classes du package devront adopter.

Cette interface est définie comme suit :

```
package Function;

import Graph.*;

public interface Function {
    public void applyGraph(Graph graph);
    public void applyVertex(Vertex vertex);
    public void applyArc(Arc arc);
}
```

L'astuce est que chaque élément de la structure de données, c'est-à-dire le graphe, les nœuds et les arcs, a une méthode *apply*.

Dans le cas d'un nœud, la classe qui le représente (la classe *Vertex* définie dans la section 3.4 Implémentation) contient une méthode *apply* qui est définie comme suit :

```
public void apply(Function f) {
    f.applyVertex(this);
}
```

Cette méthode reçoit en argument un objet de type *Function* qui, dans notre cas, sera un objet qui implémente l'interface *Function*, c'est-à-dire un des algorithmes de calcul. La méthode *apply* du nœud fait alors appel à la méthode *applyVertex* de l'objet reçu en argument.

Par exemple, lors du calcul des niveaux de dessin, nous commençons par instancier un objet de type *FComputeLevels*. La classe *FComputeLevels* contient l'algorithme qui a pour but de calculer le niveau de dessin de chacun des nœuds du graphe. Cette classe implémente l'interface *Function*. Ensuite, nous appelons la méthode *apply* du graphe, la référence du graphe étant connue. Cette méthode appelle la méthode *applyGraph* de l'objet reçu en argument, c'est-à-dire de l'objet de type *FComputeLevels*. La méthode *applyGraph* est définie de manière à parcourir la séquence des nœuds du graphe et à appliquer à chacun d'eux la méthode *apply* de la classe *Vertex*. Le même système est utilisé pour les nœuds et pour les arcs.

La première étape est donc :

```
FComputeLevels fComputeLevels = new FComputeLevels(graph.getNbrVertices());
graph.apply((Function)fComputeLevels);
```

La classe *FComputeLevels* est définie comme suit :

```
package Function;

import GUI.*;
import Graph.*;
import GIterator.*;

public class FComputeLevels implements Function{
    private int _cptDepth = 0;
    private boolean[] _visitDfs;

    public FComputeLevels(int nbrVertices) {
        _visitDfs = new boolean[nbrVertices];
        for(int j=0; j<_visitDfs.length;j++) _visitDfs[j] = false;
    }
    public void applyVertex(Vertex vertex) {
        if(_visitDfs[vertex.getId()] == false){
            _visitDfs[vertex.getId()] = true;
            GraphicDecorator dec =
                                (GraphicDecorator)vertex.getDecorator();
            dec.setLevel(_cptDepth);
            GIterator it = vertex.getDescendents();
            it.reset();
            while (it.hasMore()) {
                _cptDepth++;
                Arc arc = (Arc)it.getCurrent();
                arc.apply(this);
                _cptDepth--;
                it.next();
            }
        }
    }
    public void applyArc(Arc arc) {arc.getSink().apply(this);}
    public void applyGraph(Graph graph) {
        GIterator it = graph.getVertices();
        it.reset();
        while(it.hasMore()) {
            Vertex v = (VertexI)it.getCurrent();
            v.apply(this);
            it.next();
        }
    }
}
```

Ce système permet ainsi de définir une fois pour toute l'arbre suivant lequel le graphe sera parcouru et de lui appliquer n'importe lequel des algorithmes définis dans une classe qui implémente l'interface *Function*.

Cette méthode facilite la tâche du programmeur en établissant un canevas commun à tous les algorithmes. Elle permet également une meilleure lisibilité du code source.

3.4 L'IMPLÉMENTATION

3.4.1 Les structures de données

La représentation du graphe nécessite l'utilisation de trois types d'objets (ou classes) : *Graph* représentant le graphe, *Vertex* représentant un nœud et *Arc* représentant un arc. Cependant, comme nous ne sommes pas encore certains de la manière dont nous allons implémenter ces éléments, nous avons décidé de leur associer à chacun une interface. Ainsi, nous aurons un canevas précis de ce qui doit être implémenter sans nécessairement imposer une implémentation particulière.

Nous avons donc trois interfaces, dont les noms sont ceux cités ci-dessus, et trois classes qui implémentent ces interfaces : *GraphI*, *NœudI* et *ArcI*. Le *I* ajouté à la fin de ces noms signifie qu'il s'agit de l'implémentation de l'interface correspondante.

En plus de cela, deux classes sont nécessaires à la représentation d'une séquence de *Vertex* et d'une séquence d'*Arc*. Ces classes ont été arbitrairement appelées *ArrayVertices* et *ArrayArcs*, indépendamment du fait que ces séquences soient implémentées par une table, une liste ou tout autre structure. A chacune de ces séquences est associé un itérateur calqué sur l'interface *GIterator*. Pour la classe *ArrayVertices*, nous utilisons la classe *GIteratorV*, et, pour la classe *ArrayArcs*, nous utilisons la classe *GIteratorA*.

Afin de rendre le programme le plus évolutif possible, nous avons décidé, dès le départ, de séparer les éléments représentant le graphe (*Graph*, *Vertex* et *Arc*) des éléments permettant son affichage. Ces derniers sont étroitement liés à sa structure mais sont malgré tout dissociés de celle-ci.

En effet, à chaque nœud est associé un décorateur. En toute généralité, le décorateur peut se voir attribuer toutes sortes de tâches telles que l'impression, le dessin à l'écran, des calculs divers ... Dans notre cas, il s'agit d'associer à chaque nœud un décorateur qui a pour tâche de dessiner le nœud et les arcs dont il est l'origine. Nous utilisons l'interface *Decorator* afin de définir un modèle de base pour les décorateurs. En ce qui concerne le dessin, ce sont les objets de la classe *GraphicDecorator* qui sont sollicités. Il n'y a actuellement pas d'autre décorateur. Une méthode permettant d'obtenir ce décorateur est associée à chaque nœud.

Le décorateur est donc l'objet qui contient toutes les informations permettant de dessiner la boîte représentant le nœud et les arcs dont ce nœud est l'origine.

En effet, chaque décorateur fait référence à une "instance de boîte" (*InstanceBox*) qui est l'objet correspondant à ce que nous avons appelé "boîte" dans le chapitre 2. Une instance de boîte comprend les champs et les méthodes nécessaires au dessin d'un nœud.

L'instance de boîte fait référence aux "instances d'arc" (*InstanceArc*) des arcs dont le nœud est l'origine. Une "instance d'arc" contient les champs et les méthodes permettant le dessin d'un arc.

Les instances d'arc référencées par une instance de boîte forment une séquence appelée *ArrayInstanceArc*, indépendamment du fait que cette séquence soit implémentée par une table, une liste ou tout autre structure. Cet ensemble est du même type que les séquences précédemment nommées et a également un itérateur attitré, *GIteratorIA*.

Chacune de ces classes contient des attributs, c'est-à-dire des informations particulières la concernant et des méthodes permettant d'interagir avec les objets de la classe en question. Nous renvoyons le lecteur à l'annexe 2 pour plus de renseignements à ce sujet.

3.4.2 *L'architecture*

L'architecture de l'utilitaire d'affichage peut être exprimée en termes de packages. En effet, un package est associé à chaque ensemble de classes et d'interfaces qui touchent à un même aspect.

Il y a quatre packages : *Iterator* qui contient l'interface *GIterator*, *Graph* qui englobe les classes et interfaces des structures de données, *GUI* qui est l'ensemble des classes permettant le dessin du graphe et *Function* qui rassemble toutes les classes qui correspondent chacune à un des algorithmes de calculs décrits plus haut.

Chacun de ces packages regroupe donc plusieurs classes et interfaces dont nous donnons un bref descriptif ci-dessous.

- ITERATOR

  Ce package contient uniquement l'interface GIterator.

  Interface :

GIterator	Interface représentant un itérateur.

- GRAPH

  Ce package contient toutes les classes directement liées à la notion de graphe. Il s'agit donc des structures de données permettant de représenter un graphe.

  Interfaces :

Graph	Interface représentant un graphe.
Arc	Interface représentant un arc.
Vertex	Interface représentant un nœud.

  Implémentations :

GraphI	Implémentation correspondant à l'interface *Graph*.
ArcI	Implémentation correspondant à l'interface *Arc*.
ArrayArc	Séquence d'arcs.
GIteratorA	Implémentation correspondant à l'interface *GIterator*. Dans ce cas, l'itérateur permet de parcourir un *ArrayArc*.
VertexI	Implémentation correspondant à l'interface *Vertex*.
ArrayVertices	Séquence de nœuds.
GIteratorV	Implémentation correspondant à l'interface *GIterator*. Dans ce cas, l'itérateur permet de parcourir un *ArrayVertices*.

- GUI

  Ce package contient toutes les classes directement liées à l'interface graphique mis à part les algorithmes.

  Interface :

Decorator	Interface correspondant à l'élément graphique associé à chaque nœud.

  Implémentation :

DrawGraphFrame	Dessine la fenêtre, construit et affiche le graphe.
DrawGraphPanel	Panel dans lequel le graphe est dessiné.
ComputeGraph	Instancie les classes de calcul nécessaires au dessin du graphe.
Scales	Couple représentant l'échelle à laquelle le dessin est affiché.
GraphicDecorator	Implémentation du *Decorator*.
InstanceArc	Permet de dessiner un arc. Cet objet contient les points de routage de l'arc.
ArrayInstanceArc	Séquence des *InstanceArc* des arcs issus d'une boîte.
GIteratorIA	Implémentation correspondant à l'interface *GIterator*. Dans ce cas, l'itérateur permet de parcourir un *ArrayInstanceArc*.
InstanceBox	Permet de dessiner la boîte.
ArrowIcon	Permet de dessiner la flèche au bout d'un arc.
DrawingConstants	Constantes de dessin.
ColorNode	Couleurs associées à un nœud. Nécessaire dans certain parcours du graphe.
KindArc	Type d'un arc.

- FUNCTION

  Ce package contient tous les algorithmes nécessaires au dessin du graphe représentant l'environnement et le store.

  Interface :

Function	Interface présentant la structure générale de toutes les classes de calcul.

  Implémentation :

  (Instanciée dans DrawGraphFrame)

FDfsPrint	Imprime le graphe en utilisant un algorithme DFS.
FPaint	Dessine le graphe en appelant la fonction *paint* du *Decorator*.

FFitToWindow	Zoom (en positif ou en négatif) le dessin du graphe de manière à ce qu'il remplisse la fenêtre.
(Instanciée dans ComputeGraph)	
FComputeLevels	Calcule le niveau de dessin de chaque nœud.
FInitInstanceBoxes	Crée l'*InstanceBox* correspondant à chaque *Decorator*.
FComputeKind	Associe un type à chaque arc.
FDfsVisit	Détermine quels nœuds devront être dessinés en fonction de la profondeur désirée.
FComputeSpaceToAddAfter	Calcule le nombre de couloirs à ajouter après chaque boîte.
FComputeSpaceToAddBefore	Calcule le nombre de couloirs à ajouter avant chaque boîte.
FComputeSpaceToAddUnderBack	Calcule le nombre de couloirs à ajouter sous chaque *Bounding Box* afin d'avoir la place pour faire passer les arcs de type BACK.
FComputeSpaceToAddUnderCross	Calcule le nombre de couloirs à ajouter sous chaque *Bounding Box* afin d'avoir la place pour faire passer les arcs de type CROSS.
FComputeMaxSpaceToAdd	Calcule le nombre de couloirs maximal à ajouter avant et après les boîtes de chaque niveau.
FComputeHeightBox	Calcule la hauteur de chaque boîte.
FComputeHeightBBox	Calcule la hauteur de chaque Bounding Box.
FComputeWidthBBox	Calcule la largeur de chaque Bounding Box.
FComputeCoordBBox	Calcule les coordonnées de chaque Bounding Box.
FComputeBBox	Calcule la hauteur et la largeur de la *Bounding Box* qui contient toutes les composantes simplement connexes du graphe.
FComputeAnchor	Calcule le point d'encrage de chaque boîte.
FComputeRouting	Calcule les points de routage de chaque arc.

CHAPITRE 4

AMÉLIORATIONS À APPORTER

Ce chapitre contient la liste des objectifs qui n'ont pas encore été atteints. Cette liste se divise en plusieurs catégories : une catégorie concernant le dessin du graphe en lui-même, une catégorie visant la performance, une autre catégorie à propos des propriétés de dessins et une dernière catégorie pour l'interface de l'utilitaire.

Après avoir réalisé une première version de l'utilitaire, nous avons remarqué que plusieurs améliorations devaient encore y être apportées.

En effet, le dessin du graphe représentant l'environnement et le store ne répond pas encore à tous les objectifs qui avaient été fixés. Nous avons également remarqué quelques défauts dans l'algorithme de routage des arcs.

De plus, lorsque le nombre de nœuds est trop élevé, le temps d'attente entre l'ordre de dessin et l'affichage du graphe est beaucoup trop long.

Ensuite, nous désirons réorganiser les propriétés liées au dessin et en augmenter le nombre.

Enfin, l'interface de l'utilitaire n'est pas encore au point.

4.1 DESSIN DU GRAPHE

Tout d'abord, dans le but de faciliter la lecture du graphe, nous avons décidé que trois points de destination seraient prévus pour l'extrémité finale des arcs, une pour les arcs d'arbre, une pour les arcs arrières et une pour les arcs transversaux.

Ensuite, le critère esthétique que nous avons décidé de respecter lors du choix des paramètres de dessin du graphe représentant l'environnement et le store n'est pas atteint. Bien que l'algorithme de routage des arcs tend à répondre à celui-ci, il ne nous permet pas encore d'assurer que le nombre de croisements d'arcs est minimal. Afin de pouvoir affirmer que ce critère est respecté, il est nécessaire de changer l'ordre selon lequel les arcs sont dessinés devant et derrière chaque boîte. Une première tentative a été réalisée mais elle n'est pas encore au point.

De plus, nous avons remarqué que, devant et derrière chaque boîte, les arcs se situent parfois à une distance trop grande. Nous devons donc faire en sorte que cette distance soit réduite en testant, pour chaque nœud, si le couloir par lequel nous voulons le faire passer est occupé ou non.

Enfin, la contrainte selon laquelle les nœuds faisant partie de l'environnement doivent être la racine d'un arbre et être rassemblés sur le côté gauche du dessin n'est pas encore respectée. En effet, nous avons réalisé que les boîtes ne permettent pas de représenter un élément de l'environnement. Cette modification nous semble simple mais elle reste à faire.

Une dernière remarque concerne un éventuel test de planarité. En effet, les graphes planaires peuvent être dessinés de manière à respecter parfaitement le critère esthétique concernant le croisement des arcs. Il serait donc intéressant de déterminer si le graphe à dessiner est planaire ou non et de choisir l'algorithme de dessin qui convient le mieux à la situation rencontrée.

4.2 PERFORMANCE

Lorsque le nombre de nœuds est fort élevé, le temps d'attente avant l'affichage du graphe est beaucoup trop long. Pour remédier à ce problème, nous devons utiliser la classe DubbleBuffering.

4.3 PROPRIÉTÉS DE DESSIN

Nous disposons de zooms permettant d'agrandir ou de réduire le dessin du graphe. Cependant, nous désirons permettre à l'utilisateur de faire des zooms locaux, c'est-à-dire d'agrandir une partie du dessin. Cette partie doit pouvoir être sélectionnée en cliquant et en faisant glisser la souris de manière à entourer un rectangle. C'est ce rectangle qui fera l'objet d'un agrandissement.

Nous souhaitons également permettre de dessiner les arcs d'arbre parallèlement aux côtés des boîtes.

Enfin, nous désirons assigner aux arcs des couleurs différentes selon leur type et permettre à l'utilisateur de choisir ces couleurs via des *comboBoxes*.

Les autres propriétés doivent également faire l'objet de modifications quant à leur présentation et à leur accès via l'interface.

4.4 INTERFACE DE L'UTILITAIRE

Comme la largeur assignée aux boîtes est fixée, il se peut que des noms d'objet ou d'attribut ne puissent être affichés en entier. Afin de permettre à l'utilisateur de connaître le nom d'un objet ou d'un attribut dont l'affichage est incomplet, nous avons réalisé une méthode inscrivant ces noms dans un *tooltip*. En effet, en fonction de la position de la souris, nous savons de quel nom il s'agit et nous l'affichons dans un *tooltip*. Cependant, cet affichage ne fonctionne pas encore de manière régulière.

Nous désirons également améliorer l'interface de l'utilitaire d'affichage en y ajoutant notamment une fenêtre de capture de fichiers permettant de choisir le fichier contenant le graphe à afficher. Plus d'informations sur ces fichiers sont disponibles au sein de l'annexe 1.

De plus, nous comptons ajouter les éléments permettant de rendre l'interface de l'utilitaire la plus accueillante possible. Nous souhaitons notamment lui ajouter un menu.

CONCLUSION

Ce travail a été réalisé dans le cadre du projet JavAbInt. Le but de celui-ci est de développer "un analyseur statique générique de Java par interprétation abstraite". Sa finalité est donc d'appliquer les techniques de l'interprétation abstraite au langage Java afin de réduire les problèmes d'inefficacité liés à l'héritage, au polymorphisme et plus spécialement aux liens dynamiques.

Le lien entre l'interprétation abstraite et la théorie des graphes a été abordé dès le début de ce mémoire. En effet, l'objectif de ce dernier est la réalisation d'un utilitaire permettant l'affichage de l'environnement et du store d'un programme Java. Or le store d'un programme peut être représenté sous la forme d'un graphe dirigé, les éléments de l'environnement étant des sommets particuliers du graphe. Il s'agit donc de développer une application permettant de dessiner un graphe quelconque.

Avant de rentrer dans le vif du sujet, il nous a semblé nécessaire de présenter quelques rappels concernant la théorie des graphes.

Ensuite, nous avons déterminé les paramètres qui guideraient le dessin du graphe représentant un couple constitué d'un environnement et d'un store. Nous avons ainsi choisi de respecter trois conventions portant toutes sur la position des arcs et des nœuds, un critère esthétique concernant la minimisation des croisements d'arcs et une contrainte à propos des nœuds représentant l'environnement. Ces paramètres ont été déterminés de manière subjective mais dans le but de répondre le mieux possible aux attentes des utilisateurs d'un tel outil.

Dès lors, nous avons décrit l'algorithme de parcours du graphe qui est la pierre angulaire de l'application. Les autres algorithmes nécessaires au dessin du graphe ont été définis par la suite puisqu'ils ont la même structure que l'algorithme de parcours du graphe. L'algorithme de dessin en lui-même n'a pu être présenté qu'après avoir dépeint l'ensemble de ces algorithmes.

Enfin, nous avons abordé brièvement les divers éléments qui rendent l'utilitaire plus intéressant et plus convivial.

Certains aspects de la programmation orientée objet, et plus particulièrement de Java, nous semblaient mériter quelques approfondissements. Nous avons donc consacré une partie de ce travail aux explications de ces éléments.

L'implémentation de cet utilitaire a été réalisée au cours d'un séjour de trois mois et demi à Providence, aux États-Unis. Nous y avons été accueillis par le département informatique de l'université de Brown. C'est au cours de ce stage que nous avons principalement élaboré l'utilitaire en passant par une période d'adaptation au langage Java.

Il s'agit de la deuxième version de l'utilitaire d'affichage, une première version ayant été développée au cours d'un stage réalisé par K. Noben dans le cadre de son mémoire.

Cette deuxième version de l'utilitaire d'affichage de l'environnement et du store avait pour objectif primaire de respecter l'ensemble des paramètres de dessin. Cependant, seules les conventions ont été respectées, le critère esthétique et la contrainte n'ont pas encore été atteints. En effet, en ce qui concerne le critère esthétique, des modifications plus ou moins importantes doivent être apportées à ce qui existe déjà afin de minimiser le nombre de croisements entre les arcs. Pour ce qui est de la contrainte, les éléments de l'environnement n'ont pas encore été pris en compte mais la solution ne semble pas nécessiter de recherche supplémentaire.

De plus, l'interface en elle-même nécessite quelques améliorations afin d'être plus accueillante et plus facile d'utilisation.

Bien que n'ayant pas atteint tous les objectifs fixés, nous pouvons affirmer qu'un deuxième pas a été franchi dans la mise en œuvre de l'utilitaire d'affichage de l'environnement et du store. Nous espérons que ce travail pourra être poursuivi afin d'atteindre l'ensemble de ces objectifs.

BIBLIOGRAPHIE

1. [DBETT] Giuseppe DI BATTISTA, Peter EADES, Roberto TAMASSIA et Ioannis G. TOLLIS. *GRAPH DRAWING : Algorithms for the visualization of graphs.* Prentice Hall, *Inc.*, 1999.

2. [EPPSTEIN] David EPPSTEIN. *ICS 161: Design and Analysis of Algorithms – Lecture notes for Febuary 15, 1996.* UC Irvine: Dept. Information & Computer Science, 1996. Disponible sur Internet à l'adresse www.ics.uci.edu/~eppstein/161/960215.html.

3. [FICHEFET] Jean FICHEFET. *Théorie de graphe – Notes de cours.* Facultés Universitaires Notre Dame de La Paix de Namur : Faculté d'informatique. 1999.

4. [JAVACORE] Cay S. HORSTMANN et Gary CORNELL. *Core JAVA, Volume 1 – Fundamentals.* The Sun Microsystems Press. Prentice Hall, *Inc.*, 1999.

5. [JAVADOC] *Java 2 Platform, Standard Edition, v1.2.2, API Specification.* Sun Microsystems, *Inc.*. Disponible sur Internet à l'adresse java.sun.com/products/jdk/1.2/docs/api/index.html.

6. [JAVAMAC] Alexander Newman *et al. Le Macmillan Java.* Simon & Shuster Macmillan, 1996.

7. [JAVATUT] The *Java Tutorial : A practical guide for programmers.* Sun Microsystems, *Inc.*, mai 2001. Disponible sur Internet à l'adresse java.sun.com/docs/books/tutorial.

8. [KDL] Rob KRAVITZ, Rob DAVID et Rich LAFFERTY. *Winter 1997 Class Notes for 308-251B – Data structures and algorithms – Topic#26: Depth-First Search.* McGill University: School of Computer Science, 1997. Disponible sur Internet à l'adresse www.cs.McGill.CA/~cs251/OldCoures/1997/topic26.

9. [LECHARLIER] Baudouin LE CHARLIER. *L'analyse statique des programmes par interprétation abstraite.* Facultés Universitaires Notre-Dame de La Paix de Namur : Faculté d'informatique. Nouvelle de la science et des technologies, 9(4):19-25, 1992.

10. [NELSON] Randal NELSON. *CSC 173: Computation and Formal System – Searching a Graph – Class notes.* Rochester University, 1996. Disponible sur Internet à l'adresse www.cs.rochester.edu/u/nelson/courses/csc_173/graphs/search.html.

11. [NOBEN] Karl NOBEN. *Vers un analyseur statique générique de Java par interprétation abstraite : Un analyseur statique simple et un utilitaire d'affichage de l'environnement et du store.* Facultés Universitaires Notre Dame de La Paix de Namur : Faculté d'informatique. Mémoire de fin d'étude, septembre 2000.

12. [POLLET] Isabelle POLLET. *Sémantiques opérationnelles et domaines abstraits pour l'analyse statique de Java.* Facultés Universitaires Notre-Dame de La Paix de Namur : Faculté d'informatique. Mémoire de DEA, septembre 1999.

13. [STEINDL] Christoph STEINDL. *Static Analysis of Object-Oriented Programs.* Johannes Kepler University Linz : Institute for Practical Computer Science. Lecture Notes in Computer Science, Springer Velga, 1999. Disponible sur Internet à l'adresse www.ssw.uni_linz.ac.at/Research/Papers/Ste99b.html.

ANNEXE 1

CONSTRUCTION DU GRAPHE À DESSINER

Avant de s'intéresser au dessin du graphe et aux calculs qui l'accompagnent, il nous a fallu trouver un moyen pour communiquer le graphe au programme. En effet, pour manipuler et dessiner le graphe, il est nécessaire de connaître les nœuds et les arcs qui le composent, de manière à instancier les objets qui le représentent.

A. Le fichier

Pour faciliter cette étape, nous avons décidé d'utiliser un fichier intermédiaire comme paramètre du programme. Ce fichier est constitué de *(n+1)* lignes par composante simplement connexe, *n* étant le nombre de nœuds de la composante en question. Il peut y avoir plusieurs composantes dans un même graphe. Voici la description du fichier :

- *<fichier> ::= <nombre de nœuds> <return> [<ligne> <return>]**
- *<nombre de nœuds>* $\in$ *Entier*
  Cet entier est égal au nombre de nœuds qui constituent le graphe.
- *<ligne> ::= <identifiant du nœud>-<nom du nœud>*
  | <identifiant du nœud>-<nom du nœud>-<liste des successeurs>
  Les lignes sont triées en ordre croissant sur les identifiants de nœud.
- *<identifiant du nœud>* $\in$ *Entier*
  Chaque nœud a un identifiant, distinct de tous les autres nœuds, tel que l'identifiant de nœud d'une ligne doit être égal à l'identifiant de nœud de la ligne précédente + 1, le premier identifiant de nœud du fichier étant 0. S'il y a plusieurs composantes simplement connexes, l'identifiant de nœud de la première ligne d'une composante doit être égal à l'identifiant de nœud de la dernière ligne de la composante précédente + 1, sauf s'il s'agit de la première composante décrite dans le fichier.
- *<nom du nœud> ::= <caractère>**
- *<caractère>* $\in$ *{tous les caractères} \ {-}*
- *<liste des successeurs> ::= <identifiant du nœud> [, < identifiant du nœud>]**
  La liste des successeurs est triée en ordre croissant sur les identifiants de nœud.

Si les "-" séparent les différents blocs d'une ligne, nous pouvons considérer que, dans ce fichier, les lignes se composent de un, deux ou trois blocs. En effet, les lignes indiquant le nombre de nœuds du graphe ne comportent qu'un seul bloc; celles décrivant un nœud sans successeur ont deux blocs ; et celles décrivant un nœud ayant des successeurs comptent trois blocs. Le troisième bloc est lui-même décomposé en sous-blocs correspondant chacun à un successeur du nœud. Ces sous-blocs sont séparés par des ",".

exemple A1.1

3 → **1 bloc**
0-Zero-1,2 → **3 blocs**
1-One → **2 blocs**
2-Two-0

C'est à partir de ce fichier, et suivant l'algorithme de construction du graphe (*ALGO1*), que les structures de données représentant le graphe vont être créées.

B. La construction du graphe

Cette construction se fait via l'intermédiaire de deux méthodes de la classe *DrawGraphFrame*. La première reçoit le fichier décrivant le graphe en paramètre et appelle pour chaque ligne du fichier la seconde méthode. Celle-ci produit les différentes structures de données représentant le graphe en guise de résultat.

La manière dont les blocs et sous-blocs sont extraits du fichier n'est pas spécifiée à ce niveau. Il s'agit de méthodes de la classe *StringTokenizer*, une classe de l'API de Java.

```
private void createGraph(String file) {
    try {
        FileReader Données  = new FileReader(new File(file));
        BufferedReader inBR = new BufferedReader(Données );

        String input;
        boolean creationOk = true;
        while ((input = inBR.readLine()) != null && creationOk == true) {
            creationOk = createTreeFromFile(input);
        }
        inBR.close();
    }
    catch (IOException ioE) {
        showError("\nFile not found.");
        _graph = null;
    }
}
```

```
private boolean createTreeFromFile(String line) {
     StringTokenizer stV = new StringTokenizer(line, "-");
     int id;
     while (stV.hasMoreTokens()) {
          switch (stV.countTokens()) {
               case 1 :
                    int nbrVertices = Integer.parseInt(stV.nextToken());
                    if (nbrVertices == 0) _graph = null;
                    else _graph = new GraphI(nbrVertices);
                    break;
               case 2 :
                    id = Integer.parseInt(stV.nextToken());
                    try {
                         Vertex source = _graph.getVertex(id);
                         source.setName(stV.nextToken());
                    }
                    catch (ArrayIndexOutOfBoundsException aoobe){
                         showError("\nError Données  file.");
                         _graph = null;
                         return false;
                    }
                    break;
               case 3 :
                    id = Integer.parseInt(stV.nextToken());
                    try {
                         Vertex source = _graph.getVertex(id);
                         source.setName(stV.nextToken());
                         StringTokenizer stA =
                              new StringTokenizer(stV.nextToken(), ",");
                         while (stA.hasMoreTokens()) {
                              int idP = Integer.parseInt(stA.nextToken());
                              Vertex sink = _graph.getVertex(idP);
                              Arc arc = _graph.addArc(source,sink);
                         }
                    } catch (ArrayIndexOutOfBoundsException aoobe){
                         showError("\nError Données  file.");
                         _graph = null;
                         return false;
                    }
                    break;
               default :
                    showError("\nError Données  file.");
                    _graph = null;
                    return false;
          }
     }
     return true;
}
```

Exemple A1.2

- *Fichier*

```
15
0-Zero-1,2,5,8
1-One-2,3
2-Two
3-Three-4
4-Four
5-Five-0,6
  6-Six-7
7-Seven-6
8-Eight-6,9,12,13
9-Nine-10,11
10-Ten-9
11-Eleven-10
12-Twelve
13-Thirteen-14
14-Fourteen-12
```

- *Séquences construites par l'algorithme*

Séquence de nœuds : [(identifiant; nom; liste ascendants; liste descendants), ...]

[(0;Zero;5;1,2,5,8), (1;One;0;2,3), (2;Two;0,1;), (3;Two;1;4), (4;Two;3;), (5;Two;0;0,6), (6;Two; 5,7,8;7), (7;Two;6;6), (8;Two;0;6,9,12,13), (9;Two;8,10;10,11), (10;Two;9,11;9), (11;Two;9;10), (12;Two;8,14;), (13;Two;8;14), (14;Two;13;12)]

Séquence d'arcs : [(origine; destination), ...]

[(0;1), (0;2), (0;5), (0;8), (1;2), (1;3), (3;4), (5;0), (5;6), (6;7), (7;6), (8;6), (8;9), (8;12), (8;13), (9;10), (9;11), (10;9), (11;10), (13;14), (14;12)]

- *Schéma*

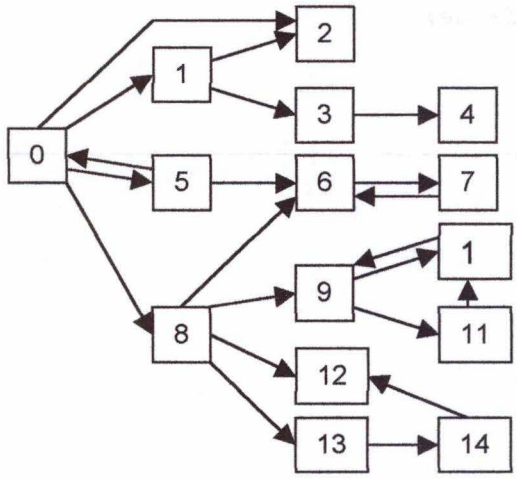


Figure A1.1 : Le graphe construit.

ANNEXE 2

STRUCTURES DE DONNÉES

Les spécifications qui suivent ont pour but de permettre au lecteur d'avoir une idée plus précise de la manière dont les structures de données ont été implémentées. Cependant, ces spécifications ne sont accompagnées d'aucun code Java.

Toutes les classes dont nous avons parlé dans la présentation des structures de données au chapitre 3 sont présentées. Cependant, comme les classes implémentant l'interface *GIterator* sont pratiquement identiques, nous nous sommes limités à l'explication de l'une d'entre elle, *GIteratorV*.

Nous noterons que les coordonnées d'un point peuvent être représentées par une classe propre à Java dont le nom est *Point*, et que la *Bounding Box* associée à un nœud est représentée par un rectangle, d'où l'utilisation de la classe *Rectangle*, propre à Java. Ces éléments apparaîtront dans la suite de la description des éléments de la structure de données.

A. Notations

Par soucis de clarté, voici les notations qui seront utilisées pour présenter la spécification des classes et des méthodes constituant ces classes.

Pour la spécification d'une **classe**, la notation suivante sera dorénavant utilisée :

- *Attributs :*

  Une simple énumération des attributs de la classe.

- *Méthodes :*

  La spécification de chaque méthode de la classe en utilisant pour chacune d'elles la notation décrite ci-après.

Pour la spécification d'une **méthode** de classe, la notation suivante sera dorénavant utilisée :

nomMéthode	*Données*	Section contenant les paramètres de la méthode. Nous noterons l'absence de paramètres par */*.
	Résultats	Section contenant le résultat de la méthode s'il s'agit d'une Fonction. Dans le cas d'une procédure, nous noterons l'absence de résultat par */*.
	Fonction	Section contenant la description de l'utilité de la méthode.

Le constructeur d'une classe sera dénoté par le nom de la classe.

B. GIteratorV

- *Attributs :*

Une séquence de *Vertices* et un entier représentant l'indice de l'élément courant.

- *Méthodes :*

GIteratorV	*Données*	Une séquence de *Vertices*.
	Résultats	/
	Fonction	Créer un objet de type *GIteratorV* et indicer le premier élément de la séquence.
next	*Données*	/
	Résultats	/
	Fonction	Passer à l'élément suivant l'élément courant.
previous	*Données*	/
	Résultats	/
	Fonction	Retourner à l'élément précédent.
getCurrent	*Données*	/
	Résultats	Un *Objet*.
	Fonction	Obtenir l'élément courant, c'est-à-dire l'élément désigné par l'indice au moment de l'appel de la méthode.
hasMore	*Données*	/
	Résultats	Un *Booléen*.
	Fonction	Vérifie s'il y a un élément après l'élément courant.
hasLess	*Données*	/
	Résultats	Un *Booléen*.
	Fonction	Vérifie s'il y a un élément avant l'élément courant.
reset	*Données*	/
	Résultats	/
	Fonction	Indicer le premier élément de la séquence.
last	*Données*	/
	Résultats	/
	Fonction	Indicer le dernier élément de la séquence.
add	*Données*	Un objet (*Objet*).
	Résultats	/
	Fonction	Ajouter un élément du type *Vertex* à la séquence.

C. GraphI

- *Attributs :*

Un ensemble de tous les nœuds le composant et un ensemble de tous les arcs existant entre ces nœuds.

- *Méthodes :*

GraphI	*Données*	Le nombre de nœuds du graphe (*Entier*) – N.
	Résultats	/
	Fonction	Créer un objet du type *Graph* avec un ensemble de N nœuds et un ensemble vide d'arcs.
addVertex	*Données*	L'identifiant du nœud (*Entier*) et son nom (*String*).
	Résultats	Un *Vertex*.
	Fonction	Ajouter un nœud à l'ensemble des nœuds du graphe et renvoyer ce nœud comme résultat.
addArc	*Données*	Le nœud d'origine (*Vertex*) et le nœud de destination (*Vertex*).
	Résultats	Un *Arc*.
	Fonction	Ajouter un arc à l'ensemble des arcs du graphe et renvoyer cet arc comme résultat.
getVertices	*Données*	/
	Résultats	Un *GIterator*.
	Fonction	Obtenir un itérateur sur tous les nœuds du graphe.
getNbrVertices	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir le nombre de nœuds composant le graphe.
getArcs	*Données*	/
	Résultats	Un *GIterator*.
	Fonction	Obtenir un itérateur sur tous les arcs du graphe.
getNbrArcs	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir le nombre d'arcs composant le graphe.
getVertex	*Données*	L'identifiant du nœud recherché (*Entier*).
	Résultats	Un *Vertex*.
	Fonction	Obtenir un nœud.

hasArc	*Données*	Le nœud d'origine (*Vertex*) et le nœud de destination (*Vertex*).
	Résultats	Un *Booléen*.
	Fonction	Vérifier si l'arc reliant les nœuds entrés en paramètre fait partie du graphe.

D. VertexI

- *Attributs :*

Un identifiant, un nom, ainsi que la séquence de ses prédécesseurs et la séquence de ses successeurs, un décorateur.

- *Méthodes :*

VertexI	*Données*	L'identifiant du nœud (*Entier*) (et son nom (*String*)).
	Résultats	/
	Fonction	Créer un objet du type *Vertex* avec un ensemble vide de prédécesseurs et un ensemble vide de successeurs, et lui assigner l'identifiant (et le nom) reçu(s) en paramètre.
getId	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir l'identifiant du nœud.
getName	*Données*	/
	Résultats	Une *String*.
	Fonction	Obtenir le nom du nœud.
setName	*Données*	Le nom du nœud (*String*).
	Résultats	/
	Fonction	Assigner un nom au nœud.
getInDegree	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir le nombre d'arcs dont le nœud est l'extrémité terminale.

getOutDegree correspond à ***getInDegree***, mais pour les arcs dont le nœud est l'origine.

getIncomings	*Données*	/
	Résultats	Un *GIterator*.
	Fonction	Obtenir un itérateur sur l'ensemble des prédécesseurs du nœud.

getOutgoings correspond à ***getIncomings***, mais pour les successeurs du nœud.

addIncoming	*Données*	Un arc (*Arc*).
	Résultats	/
	Fonction	Ajouter un arc à l'ensemble des prédécesseurs du nœud.

addOutgoing correspond à ***addIncoming***, mais pour les successeurs.

compareTo	*Données*	Un nœud (*Vertex*).
	Résultats	Un *Booléen*.
	Fonction	Vérifier si le nœud reçu en argument est le même nœud que le nœud courant.
getDecorator	*Données*	/
	Résultats	Un *GraphicDecorator*.
	Fonction	Obtenir le décorateur associé au nœud.

E. ArcI

- *Attributs :*

Son origine (*Source*) et sa destination (*Sink*).

- *Méthodes :*

ArcI	*Données*	Le nœud d'origine (*Vertex*) et le nœud de destination (*Vertex*).
	Résultats	/
	Fonction	Créer un objet du type *Arc* en assignant les nœuds d'origine et de destination reçus en paramètre aux attributs correspondants.
getSource	*Données*	/
	Résultats	Un *Vertex*.
	Fonction	Obtenir l'origine de l'arc.
getSink	*Données*	/
	Résultats	Un *Vertex*.
	Fonction	Obtenir l'extrémité terminale de l'arc.
compareTo	*Données*	Un nœud (*Vertex*).
	Résultats	Un *Booléen*.
	Fonction	Comparer deux arcs et vérifier s'il s'agit du même arc ou pas.

F. ArrayVertices

- *Attribut :*

Une séquence de nœuds.

- *Méthodes :*

ArrayVertices	*Données*	Le nombre de nœuds appartenant à la séquence de nœuds (*Entier*) – N.
	Résultats	/
	Fonction	Créer une séquence de N nœuds.
add	*Données*	Un nœud (*Vertex*).
	Résultats	/
	Fonction	Ajouter un nœud à la séquence.
add	*Données*	Un nœud (*Vertex*) et un indice (*Entier*).
	Résultats	/
	Fonction	Ajouter un nœud à l'indice spécifié.
get	*Données*	Un indice (*Entier*).
	Résultats	Un *Vertex*.
	Fonction	Obtenir le nœud correspondant à l'indice spécifié.
size	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir la taille de l'ensemble.
isEmpty	*Données*	/
	Résultats	Un *Booléen*.
	Fonction	Vérifier si la séquence est vide ou pas.
getIterator	*Données*	/
	Résultats	Un *GIterator*.
	Fonction	Obtenir un itérateur sur la séquence.

G. ArrayArcs

- *Attribut :*

Une séquence d'arcs.

- *Méthodes :*

ArrayArcs	*Données*	/
	Résultats	/
	Fonction	Créer une séquence vide d'arcs.
add	*Données*	Un arc.
	Résultats	/
	Fonction	Ajouter un arc à la séquence.

get	*Données*	Un indice (*Entier*).
	Résultats	Un *Arc*.
	Fonction	Obtenir l'arc correspondant à l'indice spécifié.

Les méthodes ***size***, ***isEmpty*** et ***getIterator*** sont identiques à celle de *ArrayVertives*.

H. GraphicDecorator

- *Attributs :*

Le niveau de dessin du nœud, le nœud dont il est le décorateur, l'instance de boîte qui lui est associée, la *Bounding Box* qui lui correspond, le point d'encrage de l'instance de boîte représentant le nœud.

- *Méthodes :*

GraphicDecortor	*Données*	Le nœud dont il sera le décorateur.
	Résultats	/
	Fonction	Créer un objet de type *GraphicDecorator* et associer le nœud reçu en argument avec l'attribut qui lui correspond.
getInstanceBox	*Données*	/
	Résultats	Une *InstanceBox*.
	Fonction	Obtenir l'instance de boîte associée au décorateur du nœud.
setInstanceBox	*Données*	L'instance de boîte correspondant au nœud (*InstanceBox).*
	Résultats	/
	Fonction	Assigner une instance de boîte au décorateur du nœud.
getVertex	*Données*	/
	Résultats	Un *Vertex*.
	Fonction	Obtenir le nœud correspondant au décorateur.
getCoordBBox	*Données*	/
	Résultats	Un *Point*.
	Fonction	Obtenir les coordonnées du coin supérieur gauche de la *Bounding Box* du nœud correspondant au décorateur.
getBBox	*Données*	/
	Résultats	Un *Rectangle*.
	Fonction	Obtenir la *Bounding Box* du nœud correspondant au décorateur.

getAnchor

Données /

Résultats Un *Point*.

Fonction Obtenir le point d'encrage de la boîte associée au nœud.

setAnchor

Données Les coordonnées de la boîte (*Point*).

Résultats /

Fonction Associer les coordonnées reçues en paramètre au point d'encrage de la boîte associée au nœud.

getLevel

Données /

Résultats Un *Entier*.

Fonction Obtenir le niveau de dessin du nœud.

setLevel

Données Le niveau de dessin du nœud (*Entier*).

Résultats /

Fonction Associer le niveau reçu en paramètre avec l'attribut qui lui correspond.

I. InstanceBox

- *Attributs :*

La hauteur de la boîte, la largeur de la boîte (dont la valeur est fixe), l'espace à ajouter avant la boîte, l'espace à ajouter après la boîte, l'espace à ajouter sous la Bounding Box, l'espace maximal à ajouter avant la boîte, l'espace maximal à ajouter après la boîte, une liste de couples (arc, nœud) permettant de déterminer le nœud sous la boîte duquel le segment horizontal d'un arc (dont le nœud correspondant à l'instance de boîte est l'origine) va passer, une séquence d'arcs contenant les arcs dont le segment horizontal passe sous la boîte (les indices sont assignés aux éléments selon l'algorithme de parcours du graphe), le nœud représenté par la boîte, le décorateur de ce nœud, le point d'encrage de la boîte et l'ensemble des instances d'arcs dont le nœud (représenté par la boîte) est l'origine.

- *Méthodes :*

InstanceBox

Données Le nœud que l'instance de boîte représente (*Vertex*) et son décorateur (*GraphicDecorator*).

Résultats /

Fonction Créer un objet de type *InstanceBox* et associer le nœud et le décorateur reçus en argument avec les attributs qui leur correspondent.

getHeightBox

Données /

Résultats Un *Entier*.

Fonction Obtenir la hauteur de la boîte.

setHeightBox

Données La hauteur de la boîte (*Entier*).

Résultats /

Fonction Assigner la hauteur de boîte reçue en argument à l'attribut correspondant.

getWidthBox

Données /

Résultats Un *Entier*.

Fonction Obtenir la largeur de la boîte.

Les méthodes suivantes servent au calcul des couloirs nécessaires au dessin des arcs.

getSpaceToAddBeforeBox

Données /

Résultats Un *Entier*.

Fonction Obtenir le nombre d'espaces à ajouter avant la boîte.

Les méthodes ***getSpaceToAddAfterBox*** et ***getSpaceToAddUnderBBox*** sont semblables à la méthode ***getSpaceToAddBeforeBox***, à cela près qu'elles concernent les espaces à ajouter, respectivement, après la boîte et sous la Bounding Box.

setSpaceToAddBeforeBox

Données Le nombre d'espaces à ajouter avant la boîte (*Entier*).

Résultats /

Fonction Assigner le nombre d'espaces reçus en argument à l'attribut correspondant.

Les méthodes ***setSpaceToAddAfterBox*** et ***setSpaceToAddUnderBBox*** sont semblables à la méthode ***setSpaceToAddBeforeBox***, à cela près qu'elles concernent les espaces à ajouter, respectivement, après la boîte et sous la Bounding Box.

Les méthodes ***getMaxSpaceToAddBeforeBox***, ***setMaxSpaceToAddBeforeBox***, ***getMaxSpaceToAddAfterBox*** et ***setMaxSpaceToAddAfterBox*** ont la même spécification que les méthodes décrites ci-dessus, mis à part qu'elles portent sur le nombre total d'espaces à ajouter pour chaque niveau de dessin.

getInstancesArc

Données /

Résultats Un *ArrayInstanceArc*.

Fonction Obtenir l'ensemble des instances d'arcs dont le nœud est l'origine.

getInstanceArc

Données L'arc dont on recherche l'instance de boîte (*Arc*).

Résultats Un *InstanceArc*.

Fonction Obtenir l'instance d'arc correspondant à celui reçu en paramètre.

Les méthodes suivantes sont utilisées lors du routage des arcs.

getVertexCrossUnder

Données L'arc (*Arc*) pour lequel on désire connaître le nœud sous la boîte duquel passe le segment horizontale.

Résultats Un *Vertex*.

Fonction Obtenir le nœud sous la boîte duquel passe le segment horizontal de l'arc reçu en argument.

setVertexCrossUnder

Données Un arc (*Arc*) et le nœud (*Vertex*) sous la boîte duquel passe le segment horizontal de l'arc.

Résultats /

Fonction Ajouter un couple, formé de l'arc et du nœud reçus en argument, à la liste permettant de connaître le nœud sous la boîte duquel passe le segment horizontal de l'arc.

getArcUnder

Données Un indice (*Entier*).

Résultats Un *Arc*.

Fonction Obtenir l'élément correspondant à l'indice dans la séquence d'arcs contenant les arcs dont le segment horizontal passe sous la boîte.

setArcUnder

Données Un arc (*Arc*) dont le segment horizontal passe sous la boîte.

Résultats /

Fonction Ajouter l'arc reçu en argument à la séquence contenant les arcs dont le segment horizontal passe sous la boîte.

getArcUnderSize

Données /

Résultats Un *Entier*.

Fonction Obtenir le nombre d'éléments de la séquence contenant les arcs dont le segment horizontal passe sous la boîte.

J. InstanceArc

- *Attributs :*

L'arc représenté, le nœud d'origine de l'arc représenté, le nœud de destination de l'arc représenté, un numéro correspondant à la place de l'arc parmi les autres arcs partant du même nœud, une séquence de coordonnées correspondant aux points de routage de l'arc, le type d'arc.

- *Méthodes :*

InstanceArc	*Données*	L'arc (*Arc*) représenté et le numéro (*Entier*) correspondant à la place de l'arc parmi les autres arcs partant du même nœud.
	Résultats	/
	Fonction	Créer un objet de type *InstanceArc* et associer l'arc et le numéro reçus en argument avec les attributs qui leur correspondent.
getArc	*Données*	/
	Résultats	Un *Arc*.
	Fonction	Obtenir l'arc dont l'instance d'arc est la représentation.
getKind	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir le type d'arc correspondant à l'arc représenté. L'entier reçu en résultat correspond à un type prédéfini.
setKind	*Données*	Le type d'arc (*Entier*).
	Résultats	/
	Fonction	Assigner le type d'arc reçu en argument à l'attribut correspondant.
getNoSon	*Données*	/
	Résultats	Un *Entier*.
	Fonction	Obtenir le numéro de l'arc correspondant à la place de l'arc parmi la liste des arcs ayant la même origine.
setSizeRoutingPoint	*Données*	Le nombre de points de routage nécessaires au dessin de l'arc représenté.
	Résultats	/
	Fonction	Déterminer la taille de la liste des points de routage.

setRoutingPoint *Données* Le numéro du point de routage (*Entier*) et la coordonnée qui lui correspond (*Point*).

Résultats /

Fonction Assigner la coordonnée reçue en paramètre au point de routage correspondant au numéro reçu en argument.

K. ArrayInstanceArc

- *Attributs :*

La séquence d'instance d'arc.

- *Méthodes :*

ArrayInstanceArc *Données* /

Résultats /

Fonction Créer une séquence de N nœuds.

Les méthodes ***add***, ***get***, ***size***, ***isEmpty*** et ***getIterator*** sont semblables à celles de ArrayVertices et ArrayArcs.